UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

FILE CORRUPTION REPAIR USING REED-SOLOMON CODES

Absolvent

Theodor Negrescu

Coordonator științific

Cristian Rusu

București, iunie-iulie 2025

Abstract

The aim of this project is to implement file corruption detection and repair using Reed-Solomon error-correcting codes.

The code used is an erasure code over the field GF(2^{64}), implemented using $O(n \log n)$ transforms in a polynomial basis introduced by Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han in [3].

Finite field multiplication is implemented using carry-less multiplication, also known as XOR multiplication, and division is implemented using the extended Euclidean algorithm.

The implementation is a command-line utility which can generate parity data for a file, and later detect and repair corruption in that file.

The file is split into N blocks, and an arbitrary number of parity blocks M can be generated, for a total of $N + M$ blocks. Any N blocks are sufficient to recover the original data, so up to M corrupted blocks can be repaired.

As erasure codes require known error locations - the term 'erasure' refers to an error at a known location - a hash of each block is stored in the file header to detect corruption.

The file is not a single Reed-Solomon code, as that would require reading the entire file into memory, limiting the maximum file size which can be processed. Instead, the file is interpreted as a matrix with $N + M$ rows, and each column is an separate Reed-Solomon code.

Source code available at https://github.com/theo543/rsarc

Rezumat

Acest proiect are ca scop implementarea detecției și reparării coruperii fișierelor folosind coduri Reed-Solomon de corectare a erorilor.

Codul folosit este un cod de ștergere peste corpul GF(2^{64}), implementat folosind transformări în timp $O(n \log n)$ într-o bază polinomială introdusă de Sian-Jheng Lin, Wei-Ho Chung, și Yunghsiang S. Han în [3].

Înmulțirea în corp finit este implementată folosind înmulțire fără retenție ('carry-less'), uneori cunoscută ca înmulțire XOR, iar împărțirea este implementată folosind algoritmul extins al lui Euclid.

Implementarea este un utilitar de linie de comandă care poate genera date de paritate pentru un fișier și, ulterior, să detecteze și repare corupție în fișier.

Fișierul este împărțit în N blocuri, și un număr arbitrar de blocuri de paritate M pot fi generate, pentru un total de $N + M$ blocuri. Orice N blocuri sunt suficiente pentru a recupera datele originale, deci cel mult M blocuri corupte pot fi reparate.

Deoarece un cod de ștergere necesită cunoașterea locațiilor erorilor - termenul 'ștergere' înseamnă o eroare la o locație cunoscută - un hash al fiecărui bloc este stocat în antetul fișierului pentru a detecta corupție.

Fișierul nu este un singur cod Reed-Solomon, deoarece ar necesita citirea întregului fișier în memorie, limitând dimensiunea maximă de fișier care poate fi procesată. De fapt, fișierul este interpretat ca o matrice cu $N + M$ rânduri, iar fiecare coloană este un cod separat Reed-Solomon.

Cod sursă disponibil la https://github.com/theo543/rsarc

Contents

Chapter 1

Introduction

Data corruption is a significant issue in software and hardware systems, which can lead to data loss, incorrect program behavior, and system failures.

It is impossible in general to prevent errors due to uncontrolled factors such as electromagnetic interference, cosmic rays, or hardware defects, not to mention software bugs. Even data stored in memory or in CPU registers and cache will rarely be randomly corrupted, which must be accounted for in massive systems where it is statistically likely that regular errors will occur.

While in some cases only error detection is required, such as for network protocols which can resend data, in other cases full error correction is necessary, such as in cloud storage, backups, ECC RAM, RAID systems, and more.

Error detection and correction is a major area of research in the field of computer science, and many companies invest heavily in ensuring the robustness of their systems against random errors:

- Major cloud storage providers guarantee that data is stored redundantly and split across multiple locations, and that is it regularly scanned for errors.
- Server hardware often includes ECC memory, which can transparently correct certain errors in RAM, and which will crash the system instead of silently saving corrupted data when it cannot be corrected.
- Modern filesystems such as ZFS and Btrfs automatically detect block-level errors using checksums, and have built-in support for redundancy.

- RAID technology (Redundant Array of Independent Disks) can combine multiple disk drives into one logical unit, and store data redundantly such that a disk failure causes no data loss, with various configurations allowing for different tradeoffs between storage capacity and redundancy.

While merely detecting errors with checksums or hashes is easy, being able to correct them without having full copies of the data is a complex task that requires error-correcting codes (ECC).

A major family of error-correcting codes is Reed-Solomon codes, which have a wide range of applications, such as in CDs, DVDs, radio transmissions, QR codes, and more. Some configurations of RAID arrays use Reed-Solomon codes to recover from more than one simultaneous disk failure.

A less common application of error-correcting codes, which this project focuses on, is recovering errors within one file. Since disk errors often result in bad sectors, not necessarily full disk failures, it is useful to be able to repair a file with a few corrupted regions, without needing full copies of the file.

For this purpose, I implemented a multithreaded command-line utility which generates Reed-Solomon parity data to repair corruption in a file using interleaved Reed-Solomon codes, without using external libraries for the Reed-Solomon implementation. The program is written in Rust, a language well-suited for implementing fast system utilities, and which enables safe multithreading and memory management.

The file is divided into N data blocks, and the parity file can contain an arbitrary number M of parity blocks. This scheme can be viewed as RAID with $N + M$ drives, however the drives are not physical disks, but instead regions in one drive, or it can be viewed as interpreting the file as a matrix with $N + M$ rows, where each column is an independent Reed-Solomon code which can be processed independently. This has the drawback of a non-contiguous access pattern, which hurts performance when the data does not fit into memory.

Errors are detected using hashes of each block. A non-matching hash is considered a total loss of the block, similar to a disk failure. Any combination of N blocks is sufficient to recover the original data. Since a single error is enough to corrupt a block, errors clustered together have

much less impact than errors spread across many blocks, i.e. the code is better at recovering from burst errors than from uniform errors.

Since the number of blocks can be extremely large, as smaller blocks result in larger code sizes, and block sizes smaller or equal to disk sector sizes are desirable for maximum error correction capacity, efficient Reed-Solomon algorithms with sub-polynomial complexity are necessary. Typical algorithms used for RAID systems are not suitable, since they assume a small amount of drives, and generally do not support more than 256 blocks (due to a small field size).

Instead, I implemented a Reed-Solomon code which supports up to 2^{64} blocks, and most importantly allows $O(n \log n)$ FFT-like transforms which can be used for efficient encoding and decoding, by using a polynomial basis introduced in [3] over polynomials in the finite field GF(2^{64}).

The implementation also uses efficient algorithms for the finite field arithmetic which is at the root of Reed-Solomon codes, using a modern CPU feature - carry-less multiplication [1] - for fast finite field multiplication.

This thesis will outline the mathematics of finite field arithmetic and Reed-Solomon codes, the polynomial basis and transforms used for $O(n \log n)$ encoding and decoding, the high-level architecture of the implementation, and its performance characteristics.

Chapter 2

Reed-Solomon Codes and Finite Fields

2.1 Reed-Solomon Codes

Reed-Solomon codes are a well-known class of error-correcting codes used in a wide range of applications, from data storage to radio communication. They are based on polynomials over finite fields. [4]

The code used for this project is an erasure code over the field GF(2^{64}), implemented using $O(n \log n)$ algorithms introduced in [3].

The basic working principle of this type of Reed-Solomon code is the interpretation of data as values of a polynomial evaluated at points $\omega_0, \omega_1, \ldots, \omega_{k-1}$ in a finite field GF(2^n). The polynomial is interpolated and evaluated to obtain additional points, which are the parity information used to correct errors. The combination of data and parity points is called a codeword.

As there is only one polynomial of degree $k - 1$ or smaller passing through k points, any combination of at least k of the original and redundant points uniquely determines the original polynomial.

Because each point must have a different x-coordinate, the field size limits how much data and redundancy a single codeword can contain. In GF(2^{64}), the limit is effectively infinite.

An erasure code is a type of error-correcting code which requires that the locations of corrupted data are known. The code cannot be used to discover the locations of corrupted data by itself. In this case, hashes stored in the metadata of the parity file are used to determine error locations.

Other Reed-Solomon codes do locate errors without requiring hashes, but they are not used in this project, as hashes are a simpler and more efficient solution.

One common code is RS$(255, 223)$, which is used in CDs and DVDs, and uses 8-bit symbols (in the field GF(2^8)). The notation RS(n, k) denotes a code with n total symbols, with k data symbols and $n - k$ parity symbols. The code used in this project has no fixed n or k, they are specified by the user.

The code used in this project is also systematic, meaning that the original data is included in the output. Non-systematic codes do not include the original data, so the receiver must decode the received codeword to obtain the original data, even if no corruption occurred.

2.2 Finite Fields

Finite fields, also known as Galois fields, are mathematical structures which define addition, multiplication, subtraction, and division over a finite set of elements [2] (as opposed to the better-known infinite fields, such as the rationals, reals, and complex numbers).

A field must satisfy the following properties:

- Associativity of addition and multiplication: $(a+b)+c = a+(b+c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Commutativity of addition and multiplication: $a + b = b + a$ and $a \cdot b = b \cdot a$
- Additive and multiplicative identity elements: $a + 0 = a$ and $a \cdot 1 = a$
- Additive inverses: for every a, there exists $-a$ such that $a + (-a) = 0$
- Multiplicative inverses: for every $a \neq 0$, there exists a^{-1} such that $a \cdot a^{-1} = 1$
- Distributivity of multiplication over addition: $a \cdot (b + c) = a \cdot b + a \cdot c$

The theorems of polynomial mathematics used in Reed-Solomon codes only hold in a field, however standard computer arithmetic does not form a field. Typical arithmetic supported natively by CPUs is fixed-size binary arithmetic with overflow, which is equivalent to arithmetic modulo a power of 2. Modular arithmetic only forms a field with a prime modulus, so it cannot be used directly for Reed-Solomon codes.

For example, the operation $x \cdot 2$ is not invertible, as it is equivalent to a left shift, from which the most significant bit of x cannot be recovered.

Fortunately, it is possible to construct a field based on fixed-size integers, such as 64-bit

integers.

In a finite field GF(p^m), where p is a prime number and m is a positive integer, the elements are polynomials of degree $m - 1$, with coefficients in GF(p). For the GF(2^n) case, an element in the field is a polynomial with n coefficients, where each coefficient is a bit (i.e. a value in GF(2) = $\{0, 1\}$). For the purposes of this project, n is always 64, so the field is GF(2^{64}).

The notation ω_i is used to denote the integer i converted to an element of the field GF(2^{64}) by interpreting its bits as a polynomial, which is a no-op in code, as elements of GF(2^{64}) are stored as 64-bit integers.

It is important to note that these polynomials are not the same as the ones used in Reed-Solomon codes to represent data and parity information. Elements of GF(2^n) are simply n bit integers with more complex arithmetic. They are polynomials over GF(2), with n coefficients. Reed-Solomon polynomials can be arbitrarily long. They are polynomials over GF(2^{64}), with an arbitrary number of coefficients, and each coefficient is itself a polynomial over GF(2^2) with 64 coefficients.

Finite field addition is defined as polynomial addition. In a general field GF(p^m), this would be implemented as pairwise addition of the coefficients of two polynomials, modulo p.

In binary finite fields, addition is equivalent to XOR, as the coefficients are bits. Therefore, $x + x = 0$, and $x = -x$ (the field has characteristic 2).

Multiplication is defined as polynomial multiplication, followed by reduction modulo an irreducible polynomial of degree 64 (with 65 coefficients, where the highest coefficient is 1).

The irreducible polynomial used for this project is $x^{64} + x^4 + x^3 + x + 1$ [5]. The choice of irreducible polynomial does not affect correctness, and the fields obtained from different choices are isomorphic.

2.3 Efficient Finite Field Arithmetic

The efficiency of finite field arithmetic is crucial for the performance of the encoding and decoding algorithms.

Addition is simple and fast, as it is equivalent to XOR. Multiplication, however, is slower and more complicated than integer multiplication, and division even more so.

2.3.1 Russian Peasant Algorithm

The Russian peasant algorithm multiplies two values in GF(2^{64}) without requiring 128-bit integers. It incrementally performs the multiplication by adding intermediate values into an accumulator, and slowly shifting the values to be multiplied and applying polynomial reduction.

This algorithm is fairly simple and easy to implement, however multiplication can be done far more efficiently on modern CPUs using carry-less multiplication. Still, this algorithm is necessary as a fallback for older CPUs.

The state of the algorithm consists of the two values to be multiplied a and b, and an accumulator.

At each iteration, if the low bit of b is set, the accumulator is XORed with a. Then, a is shifted left, and b is shifted right.

This is justified because, at each step, we multiply the lowest coefficient of b with a, and add the result (either 0 or a) to the accumulator. Then, moving on to the next coefficient of b, we divide b by x and multiply a by x.

If the high bit of a was set before shifting, a is XORed with the irreducible polynomial, excluding the highest coefficient which would not fit in 64 bits. Conceptually, a now has a 65th bit (a coefficient x^{64}), which is removed by subtracting the irreducible polynomial.

Algorithm 1 Russian Peasant Multiplication

```
function MULTIPLY(a, b)
    acc ← 0
    for i ← 1 to 64 do
        if b & 1 then
            acc ← acc ⊕ a
        end if
        carry ← a & (1 ≪ 63)
        a ← a ≪ 1
        b ← b ≫ 1
        if carry then
            a ← a ⊕ POLYNOMIAL
        end if
    end for
    return acc
end function
```

2.3.2 Carry-less Multiplication

GF(2^{64}) multiplication can be performed using only three 128-bit carry-less multiplication operations. Modern CPUs have support for this operation, as it is useful for cryptographic algorithms, computing checksums, and other applications. [1]

The terms ”upper half” and ”lower half” will be used to refer to the most significant 64 bits and least significant 64 bits of a 128-bit integer, respectively.

By multiplying a and b using carry-less multiplication, we obtain a 128-bit result. We must reduce the upper half to a 64-bit result, which can then be XORed with the lower half to obtain the final result.

This can be done by multiplying the upper half of the result by the irreducible polynomial. Then, the lower half of the result is the product reduced modulo the irreducible polynomial.

To understand why this works, consider the process of reduction. The irreducible polynomial is aligned with each set bit in the upper half of the result, and XORed with the result. This is effectively what carry-less multiplication does.

There is a complication, however. A third multiplication is required to ensure full reduction, as the highest bits of the upper half can affect the lowest bits of the upper half.

For fields where $x^{n-1} + 1$ is irreducible, two multiplications would suffice, but this is not the case for GF(2^{64}).

For example, consider $x^{127} + x^{67} + x^{66} + x^{64}$. After aligning the irreducible polynomial with the highest bit and XORing, all bits in the upper half are zero. At this point, the reduction is complete and should stop. However, this is not how carry-less multiplication works. The irreducible polynomial will also be aligned with the other three bits, and the lower half is XORed with some unnecessary values.

The upper half of the reduced result if and where this happened. A third multiplication will correct this. The unnecessary XORs are undone by XORing with the lower half of the third multiplication.

The justification for the algorithm may seem somewhat complex, but the algorithm itself is very short, simple, and efficient.

The functions upper(x) and lower(x) return the upper and lower 64 bits of x.

Algorithm 2 Carry-less Multiplication

function MULTIPLY(a, b)
result $\leftarrow$ CLMUL(a, b)
result_partially_reduced $\leftarrow$ CLMUL(upper(result), POLYNOMIAL)
result_fully_reduced $\leftarrow$ CLMUL(upper(result_partially_reduced), POLYNOMIAL)
return lower(result) $\oplus$ lower(result_partially_reduced) $\oplus$ lower(result_fully_reduced)
end function

2.3.3 Extended Euclidean Algorithm

A simple way to compute the multiplicative inverse is to raise the element to the power of $2^{64} - 2$ using exponentiation by squaring. This was used in the early stages of this project, however it is not the most efficient method - it uses 126 multiplications. It is still used in a unit test to check the results given by the more complex algorithm.

The extended Euclidean algorithm is an extension to the well-known Euclidean algorithm for computing GCD, which also computes coefficients of Bezout's identity, a linear combination of the inputs that equals the GCD:

$$ax + by = \gcd(a, b) \tag{2.1}$$

This algorithm is the standard method for computing the multiplicative inverse in finite fields - both for integers modulo a prime and polynomials modulo an irreducible polynomial.

When setting b to the modulus of the field, $gcd(a, p) = 1$ and $py = 0$. 2.1 becomes $ax = 1$, so a is the multiplicative inverse of x modulo p.

The basic structure of the algorithm is as follows:

Algorithm 3 High-Level Extended Euclidean Algorithm

$(t, \text{new\_t}) \leftarrow (0, 1)$
$(r, \text{new\_r}) \leftarrow (p, a)$
while new_r $\neq 0$ **do**
$q \leftarrow r \div$ new_r ▷ division with remainder, not using the irreducible polynomial
$(r, \text{new\_r}) \leftarrow (\text{new\_r}, r - q \cdot \text{new\_r})$
$(t, \text{new\_t}) \leftarrow (\text{new\_t}, t - q \cdot \text{new\_t})$
end while
assert $r = 1$ ▷ t is the multiplicative inverse of a modulo p

See A.1 for details of the implementation. This algorithm, unlike the $x^{2^{64}-2}$ method, only uses around 32.5 multiplications when inverting a random element in GF(2^{64}).

Chapter 3

Polynomial Oversampling and Recovery

Standard algorithms for polynomial interpolation and evaluation, such as Newton interpolation and Horner's method, require $O(n^2)$ time. Efficient $O(n \log n)$ algorithms are used instead, based on FFT-like transforms introduced in [3].

3.1 Polynomial Basis

The polynomial basis $\mathbb{X} = \{X_0, \ldots, X_{2^{64}-1}\}$ admits transforms Ψ_h^l and $(\Psi_h^l)^{-1}$ which convert between values at h contiguous points with an arbitrary offset l and coefficients in $\mathbb{X}$, with h a power of two.

To encode a $RS(n, k)$ code, the data polynomial coefficients are obtained by applying $(\Psi_h^0)^{-1}$ to the input values, then additional values are obtained using Ψ_h^l $\frac{n}{k}$ times at offsets $l = k, 2k, \ldots, n - k$.

The basis polynomials X_i are defined as the products of polynomials $\hat{W}_j$ corresponding to the bits of the index i:

$$X_i = \prod_{j \in \text{bits}(i)} \hat{W}_j \tag{3.1}$$

$\hat{W}_i = \frac{W_i}{W_i(2^i)}$ is a normalized vanishing polynomial of degree 2^i, which vanishes (i.e. evaluates to zero) at the points $\omega_0, \omega_1, \ldots, \omega_{2^i-1}$, and evaluates to 1 at ω_{2^i}.

$$\hat{W}_i(x) = \frac{W_i(x)}{W_i(2^i)} = \frac{\prod_{j=0}^{2^i-1}(x - \omega_j)}{\prod_{j=0}^{2^i-1}(\omega_{2^i} - \omega_j)} \tag{3.2}$$

$\hat{W}_i$ has degree 2^i, as it is the product of 2^i degree one factors divided by a constant. Therefore, X_i has degree i, since it the product of W_j corresponding to the bits set in i. Since $\mathbb{X}$ contains 2^{64} polynomials with all degrees from 0 to $2^{64}-1$, it automatically is a valid basis for representing polynomials of degree up to $2^{64}-1$.

All W_i are linearized polynomials, which means they only have non-zero coefficients at power-of-two indices and are additive:

$$W_i(x+y) = W_i(x) + W_i(y) \tag{3.3}$$

Note that the standard monomial basis $\{1, x, x^2, \ldots, x^{2^{64}-1}\}$ could also be defined in a similar way, with $\hat{W}_i = X^{2^i}$, but that would not be sufficient to allow FFT-like fast transforms.

The definition of $\hat{W}_i$ as normalized vanishing polynomials is critical for simplifying equation 3.6 into 3.7, without which the transforms would not be $O(n \log n)$.

3.2 Forward and Inverse Transforms

Let D_h be the data polynomial with h coefficients $d_0, d_1, \ldots, d_{h-1}$. It can be expressed as a recursive function $\Delta_i^m(x)$, with $D_h(x) = \Delta_0^0(x)$:

$$\Delta_i^m(x) = \begin{cases} \Delta_{i+1}^m(x) + \hat{W}_i(x)\Delta_{i+1}^{m+2^i}(x) & 0 \le i \le \log_2(h) \\ d_m & i = \log_2(h) \end{cases} \tag{3.4}$$

At each step, the polynomial is split into coefficients whose index has the i-th bit set and those which don't. The final steps select the coefficient corresponding to the selected index m.

Because of the properties of the basis polynomials, the vector of $\frac{h}{2^i}$ evaluations of Δ_i^m can be computed from two vectors of size $\frac{h}{2^{i+1}}$: the evaluations of Δ_{i+1}^m and $\Delta_{i+1}^{m+2^i}$ at points with the $i+1$ least significant bits unset.

Let $\Phi(i, m, l) = [\Delta_i^m(\omega_c + \omega_l)$ for c in $[0, 2^i, \ldots, h-2^i]]$ be the vector of $\frac{h}{2^i}$ evaluations of Δ_i^m at all points $\omega_c + \omega_l$ where c has the i most significant bits unset, with l an arbitrary offset.

$\Phi(i, m, l)$ can be computed in $O(n)$ time from $\Phi(i+1, m, l)$ and $\Phi(i+1, m+2^i, l)$.

For each pair of values at index x in the two smaller vectors, the values at indices $2x$ and $2x+2^i$ in the larger vector can be computed. The values will be denoted as a, b, a', b' for clarity.

a' is straightforwardly computed as:

$$a' = \Delta_i^m(\omega_c + \omega_l) = \Delta_{i+1}^m(\omega_c + \omega_l) + \hat{W}_i(\omega_c + \omega_l)\Delta_{i+1}^{m+2^i}(\omega_c + \omega_l) = a + \hat{W}_i(\omega_c + \omega_l)b \quad (3.5)$$

The calculation of b' relies on the properties of the vanishing polynomials:

$$b' = \Delta_i^m(\omega_c + \omega_l + \omega_{2^i}) = \Delta_{i+1}^m(\omega_c + \omega_l + \omega_{2^i}) + \hat{W}_i(\omega_c + \omega_l + \omega_{2^i})\Delta_{i+1}^{m+2^i}(\omega_c + \omega_l + \omega_{2^i}) \quad (3.6)$$

The term ω_{2^i} vanishes in both Δ_{i+1}^m and $\Delta_{i+1}^{m+2^i}$, since both contain only vanishing polynomials W_j with $j \geq i+1$.

As $\hat{W}_i$ is normalized, $\hat{W}_i(\omega_c + \omega_l + \omega_{2^i}) = \hat{W}_i(\omega_c + \omega_l) + \hat{W}_i(\omega_{2^i}) = \hat{W}_i(\omega_c + \omega_l) + 1$.

$$b' = a + (\hat{W}_i(\omega_c + \omega_l) + 1)b = a + \hat{W}_i(\omega_c + \omega_l)b + b = a' + b' \quad (3.7)$$

Remember that a and b are the evaluations of Δ_{i+1}^m and $\Delta_{i+1}^{m+2^i}$ at $\omega_c + \omega_l$, whereas a' and b' evaluations of Δ_i^m at $\omega_c + \omega_l$ and $\omega_c + \omega_l + \omega_{2^i}$, respectively. While the computation of a' is simple and follows from the definition of Δ_i^m, b' is located a different index, and it is only possible to write it in terms of a and b because of the choice of basis polynomials $\hat{W}_i$.

If the standard monomial basis was used, the calculation of b' would require completely different evaluations of Δ_{i+1}^m and $\Delta_{i+1}^{m+2^i}$ at the index $\omega_c + \omega_l + \omega_{2^i}$. There would be no halving of subproblem size at each step of the recursion, so the transform would have complexity $O(n^2)$ instead of $O(n \log n)$.

Computing a and b from a' and b' is similar, and does not require finite field division:

$$b' + a' = (a' + b) + a' = b \quad (3.8)$$

$$a' + \hat{W}_i(\omega_c + \omega_l)b = (a + \hat{W}_i(\omega_c + \omega_l)b) + \hat{W}_i(\omega_c + \omega_l)b = a \quad (3.9)$$

Since the individual steps of the forward transform are reversible, the entire transform can

be run backwards to obtain the inverse transform.

The vectors can be stored interleaved in a single array, initialized to $[d_0, d_1, \ldots, d_{h-1}]$ (h single-element vectors), and then updated in-place in $log_2(h)$ steps, each step requiring $O(n)$ time.

See the butterfly diagram in [3] for a visual representation of the transforms.

In total, $n-1$ unique factors are needed - one evaluation of $\hat{W}_{\log_2(n)}$, two of $\hat{W}_{\log_2(n)-1}$, $\ldots$, $\frac{n}{2}$ evaluations of $\hat{W}_0$ - which can be computed in $O(n \log n)$ time.

See A.2 for pseudocode.

3.3 Formal Derivative

The error correction algorithm cannot directly use the inverse transform for interpolation, as the received data is not at contiguous points, and the number of non-corrupted points is likely not a power of two.

Instead, an algorithm based on the formal derivative is used which can recover the original polynomial from any k intact points regardless of error location.

In all fields, the formal derivative of a polynomial is well-defined and the standard power and product rules apply, despite the concepts of limits and continuity not existing in finite fields.

$$(f \cdot g)' = f' \cdot g + f \cdot g'$$
$$(\sum_{i=0}^{n} a_i x^i)' = \sum_{i=1}^{n} (i \cdot a_i) x^{i-1}$$

The multiplication $i \cdot a_i$ between an integer and a field element is defined as repeated addition, which in GF(2^n) is either zero or a_i, as $a_i + a_i = 0$.

$$i \cdot a_i = \begin{cases} a_i & i \text{ odd} \\ 0 & i \text{ even} \end{cases} \tag{3.10}$$

Therefore, the formal derivative of a polynomial f in GF(2^n) written in the standard mono-

mial basis is:

$$f' = a_1 + a_3x^2 + a_5x^4 + \dots \tag{3.11}$$

As the normalized vanishing polynomial $\hat{W}_i$ only has coefficients at power-of-two indices, the derivative will be a constant:

$$\hat{W}_i' = \frac{\prod_{j=1}^{2^i-1} \omega_j}{W_i(2^i)} \tag{3.12}$$

To find the derivative of the basis polynomial X_i, which is a product of up to 64 polynomials, the product rule generalized to a product of n polynomials is used:

$$(\prod_{i=0}^{n} f_i)' = \sum_{j=0}^{n} f_j' \cdot \prod_{i \neq j} f_j \tag{3.13}$$

Therefore, the derivative of X_i contains $|\text{bits}(i)|$ terms, each of which is a basis polynomial with one bit of i unset, multiplied by the derivative of the vanishing polynomial corresponding to that bit:

$$X_i' = \sum_{b \in \text{bits}(i)} \hat{W}_b' \cdot X_{i-2^b} \tag{3.14}$$

X_i' only has terms with indices less than i, so the derivative can be computed in-place by iterating from the lowest degree to the highest degree coefficients (see A.3).

The time complexity of the formal derivative is $O(n \log n)$, since at each step there are up to $\log_2(n)$ bits in the index, and computing the factors $\hat{W}_i'$ for $i = 0, 1, \dots, \log_2(n) - 1$ also takes $O(n \log n)$ time.

3.4 Polynomial Recovery

In order to recover the original polynomial using the formal derivative, an error locator polynomial is constructed which vanishes at the points where errors occurred.

Let ERASURES be the set of indices where an error occurred. As previously mentioned,

erasure codes require knowledge of all error locations, which will be obtained using hashes.

$$e = \prod_{i \in \text{ERASURES}} (x + \omega_i) \tag{3.15}$$

Since e does not depend on the actual values of the data polynomial, its values can be computed and multiplied with the incomplete data polynomial d, to zero out all unknown values.

The product rule allows the original polynomial d to be recovered:

$$(e \cdot d)' = e' \cdot d + e \cdot d' \tag{3.16}$$

$$(e \cdot d)'(\omega_x) = e'(\omega_x) \cdot d(\omega_x) + 0 \cdot d'(\omega_x) \; \forall \; x \in \text{ERASURES} \tag{3.17}$$

$$d(\omega_x) = \frac{(e \cdot d)'(\omega_x)}{e'(\omega_x)} \; \forall \; x \in \text{ERASURES} \tag{3.18}$$

Therefore, the original polynomial is recovered by multiplying d by e, applying the inverse transform, taking the formal derivative, applying the forward transform, and finally dividing by e' at the error locations.

Algorithm 4 Reed-Solomon Decoding

t_fac ← PrecomputeFactors($\log_2(n), 0$)
d_fac ← PrecomputeDerivativeFactors($\log_2(n)$)
d ← $[d_0, d_1, \ldots, d_{n-1}]$ ▷ received data
erasures ← $[i_0, i_1, \ldots, i_k]$ ▷ indices of errors
(e, e') ← ComputeErrorLocator(erasures, t_fac, d_fac)
$\hat{d} \leftarrow d \cdot e$ ▷ multiply partially corrupt data by error locator polynomial
$\hat{d}'$ ← ForwardTransform(FormalDerivative(InverseTransform($\hat{d}$, t_fac), d_fac), t_fac)
for $i \in$ erasures **do**
    $d[i] \leftarrow \hat{d}'[i]/e'[i]$
end for

For this application, (e, e') can be reused for the entire file, with e' inverted in advance, since a corrupt block causes an erasure at the same index in all the interleaved codes.

A $O(n \log n)$ algorithm for computing the error locator is described in [3] which uses fast Walsh-Hadamard transforms, however it requires 2^r operations where r is the power of the field, so it is not useful for GF(2^{64}).

Instead, I used a $O(n \log^2 n)$ recursive algorithm (A.4) which splits the polynomial into two halves, recursively computes each half, and combines the two results by multiplying in $O(n \log n)$ time using the transforms.

Chapter 4

Implementation

The implementation is written in Rust, using some third-party libraries for I/O, multithreading, hashing, and progress reporting. No libraries were used for the finite field arithmetic, polynomial operations, or Reed-Solomon codes. See A.6 for a complete list of the libraries used.

4.1 Interleaved Codes

The data and parity files are not treated as one single Reed-Solomon codeword. Instead, the data is split into blocks. The blocks are *not* independent Reed-Solomon codes, but rather the codes are split across all blocks, with each block containing one symbol from each code. In other words, the files are treated as matrices, with the blocks as rows and codes as columns.

This is necessary for several reasons:

- Using one large Reed-Solomon code would require reading all data into memory, limiting the maximum size to the available memory, or requiring complicated saving and loading to disk as part of the processing.
- The precomputed factors for one large code would be as large as the code itself, requiring large amounts of time and space to compute and store.
- Since hashes are used to detect corruption, the file must be split into blocks anyways, limiting the granularity of error correction to the block size.

Any error occurring in a block will affect all codes, as deleting a row from a matrix affects one element of each column. Multiple errors in one block have the same effect as a single error. This is suited to common error patterns, which tend not to be uniformly distributed, but instead are burst errors.

See appendix A.5 for an visual example of how burst errors are easier to correct.

4.2 Data Storage

The parity data and metadata are stored in a separate file, specified by the user.

The metadata consists of the header, which specifies the parameters of the encoding - expected data file size, number of data and parity blocks, and block size - and hashes of all blocks, used to detect corruption. The hashes also include the first 8 bytes of each block, to allow reassembly if the blocks are somehow scrambled, such as by deletion or insertion of a byte. This is very unlikely to happen, but would cause complete failure without a way to put the blocks back in order.

As the Reed-Solomon codes are split across all blocks, reading and writing the data and parity files has a very inefficient access pattern, as reading one code requires one access to each block. The blocks are analogous to rows in a matrix, and the codes to columns. Processing a matrix stored in row-major order column-wise is inherently inefficient, since non-contiguous memory access is required.

The system call overhead and seek time can be somewhat mitigated by reading many symbols per block at once, as many as can fit into memory. This produces a large buffer of interleaved symbols, which can then be processed in memory.

Reducing the length of individual codes by increasing block size improves performance, since it allows reading more symbols at once, and therefore passing through the data file fewer times. However, there is still a penalty for the non-contiguous access, especially on a hard disk drive.

The worst case scenario is if there is not enough memory to read more than one code at once, since this will require one system call per symbol. In this case, the only option is to increase the block size, which reduces code length, allowing more symbols to be read at once.

This theoretically reduces the granularity of the error correction, causing a single-byte error to render large amounts of data useless for recovery, but since burst errors are most common, this is acceptable.

The performance could also be improved by splitting a file into small sections which fit into memory, but the sections would be independent and could not be used to repair each other.

Writing is implemented using memory mapped I/O, which is simpler to use, but relies completely on the operating system to batch writes to the disk.

Since the data symbols from each code are read in batches - many codes read from each block per pass through the data file - the writes will naturally be batched as well. Testing does not show a bottleneck in writing. If necessary, the batching could be done manually, collecting output symbols in a large buffer and using normal write calls, instead of memory mapped I/O.

The same I/O code is used for both encoding and decoding. When decoding, system calls are used to read uncorrupted symbols from both files, and memory mapped I/O is used to write recovered symbols to both files. The architecture could be extended to process an arbitrary number of files, such as multiple parity files, single-file archives combining data and parity, or arbitrary folder structures as data instead of a single file.

4.3 Multithreaded Processing

To process codes in parallel, a multithreaded pipeline is used, consisting of a reader thread, an adapter thread, multiple processor threads, and a writer thread.

The reader thread reads codes into an interleaved buffer as described in the previous section. The processor threads execute the encoding or decoding algorithm, writing the output symbols into a buffer which is sent to the writer thread. The writer thread simply copies symbols from received buffers into the output memory maps, allowing the operating system to flush pages to disk asynchronously.

In order to synchronize the threads, channels are used to send messages between them. Heap-allocated buffers are used to store input and output data, moving input data from reader to adapter to processor, and output data from processor to writer.

Used input buffers are returned back to the reader and output buffers returned to the processors using separate return channels.

Filled input buffers are sent by the reader to the adapter, which creates a task message for each code in the buffer and sends it to the processor threads through a shared channel, including an offset which specifies which code to read from the buffer, and a shared atomic counter which is decremented whenever a processor thread finishes processing a code, so that when every code has been processed, the input buffer is sent back to the reader to be reused.

There are five channels used in total for the following purposes:

- Sending filled input buffers from the reader to the adapter, along with the number of codes and the index of the first code.
- Sending task messages from the adapter to the processors, containing a reference to the input buffer, a shared atomic counter, the index of the code, offset into the buffer, and number of codes in the buffer.
- Sending filled output buffers from the processors to the writer, along with the index of the code.
- Returning input buffers to the reader after every code has been processed, which is done by decrementing the shared atomic counter and returning the buffer when it reaches zero.
- Returning output buffers to the processors after the output symbols have been copied to the memory maps by the writer.

The input buffers are protected by a read-write lock, which allows multiple processors to read codes from the buffer at once, but only one thread - the reader - can write to it at a time. This is only used to ensure thread-safety, not for synchronization, which is done only using channels and the atomic counters.

The processor threads share the same precomputed factors, which depending on the task are either transform factors at multiple offsets for encoding, or transform factors plus derivative factors and the error locator polynomial for decoding.

4.4 Benchmarks

Several kinds of benchmarks were implemented to measure the performance of various aspects of the implementation - finite field arithmetic, polynomial oversampling and recovery, and the full encoding and decoding process including disk I/O.

4.4.1 Polynomial Oversampling and Recovery

This benchmark measures the performance of the encoding and decoding algorithms, as well as of the precomputation of the necessary transform factors, derivative factors, and error locator polynomial. It also includes a $O(n^2)$ algorithm from an early version of the project, which uses Newton interpolation and Horner's method, for comparison.

The results show the encoding, decoding, and precomputation algorithms scale approximately linearly with the number of symbols, as expected, with the old algorithm scaling quadratically, becoming unusable for large inputs.

For both encoding and decoding, precomputation is slower than the main algorithm, especially for recovery. However, even for a small block size of one kilobyte (equal to 128 64-bit symbols), the main algorithm will be executed over a hundred times more, so the performance of the precomputation step is not an issue.

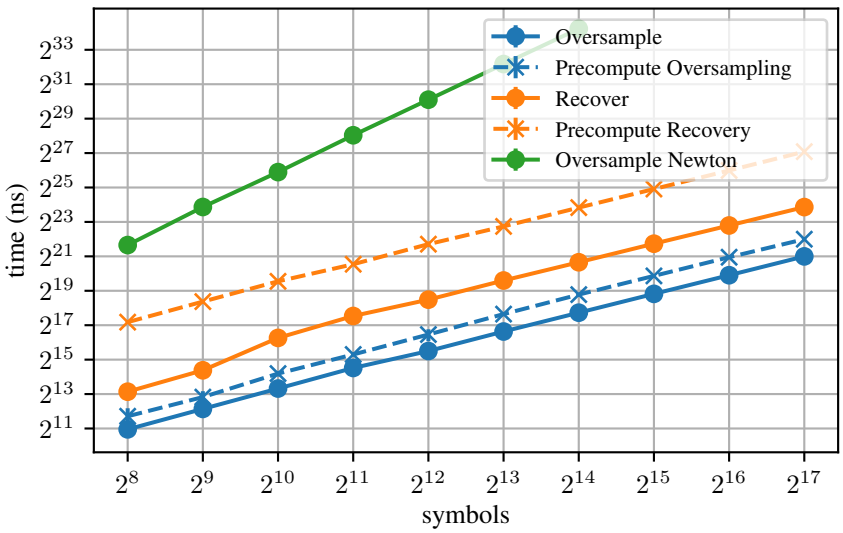

Figure 4.1: Log-scale plot of polynomial oversampling and recovery benchmarks.

4.4.2 Finite Field Arithmetic Benchmark

The results show that inversion by extended Euclidean algorithm is only slightly faster than inversion by raising to 2^{64-2}. However, with carry-less multiplication disabled, the extended Euclidean algorithm is much faster, since multiplications become more expensive. As expected, the Russian Peasant fallback is vastly slower than carry-less multiplication.

Table 4.1: Finite field arithmetic benchmark results, averaged over 2^{26} operations.

Operation	Default	No CLMUL	No CLMUL & LZCNT
Multiplication	1.32 ns	54.44 ns	58.64 ns
Inversion (Euclid)	222.96 ns	1883.14 ns	2010.98 ns
Inversion (Pow)	237.56 ns	6618.4 ns	6712.62 ns

4.4.3 End-to-End Benchmark

The full implementation was benchmarked by encoding a randomly generated file, corrupting it, and then decoding it, with 20% redundancy and a block size of 16 KB.

To detect when only I/O is being performed, CPU usage over than 15% was considered as indicating that significant computation is being performed, measured at a sample rate of 0.2 seconds. This is not a precise measurement, but it suffices to show that the program is spending most of its time not in computation, but in I/O.

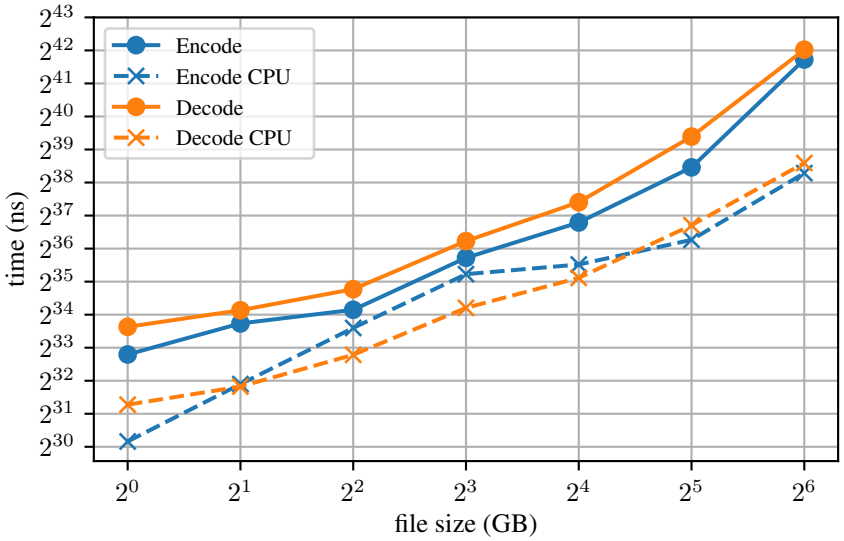

Figure 4.2: Log-scale plot of end-to-end benchmarks.

Chapter 5

Conclusions

The file metadata is currently not protected from corruption. This can be addressed by adding meta-parity blocks among the parity blocks. Although the metadata is generally small, and therefore unlikely to be corrupted, this is a significant flaw in the current implementation.

The implementation can successfully generate parity data and repair file corruption using Reed-Solomon codes in $O(n \log n)$ time.

Correctness is ensured using unit tests for the IO-free part of the implementation - the finite field arithmetic and polynomial algorithms - and end-to-end testing for the entire encoding and decoding sequence, using simulated file corruption.

Appendices

A.1 Extended Euclidean Algorithm

In the following pseudocode, leading_zeros(x) returns the number of leading zero bits in x. Modern CPUs have a dedicated instruction for counting leading zeros.

Because the irreducible polynomial is of degree 64, the first Euclidean division iteration, in the first iteration of the Euclidean algorithm, is a special case. As a 65-bit polynomial cannot fit in the 64-bit variable b, the first iteration is done manually, outside the loop.

Algorithm 5 Extended Euclidean Algorithm

```
function EXTENDEDEUCLIDEAN(a)
    assert a ≠ 0
    if a = 1 then return 1 endif
    t ← 0
    new_t ← 1
    r ← POLYNOMIAL                    ▷ the irreducible polynomial, 65-th bit excluded
    new_r ← a
    r ← r ⊕ (new_r ≪ (leading_zeros(new_r) + 1))
    quotient ← 1 ≪ (leading_zeros(new_r) + 1)
    while new_r ≠ 0 do
        while leading_zeros(new_r) >= leading_zeros(r) do
            degree_diff ← leading_zeros(new_r) − leading_zeros(r)
            r ← r ⊕ (new_r ≪ degree_diff)
            quotient ← quotient|(1 ≪ degree_diff)
        end while
        (r, new_r) ← (new_r, r)
        (t, new_t) ← (new_t, t ⊕ gf64_multiply(quotient, new_t))
        quotient ← 0
    end while
    assert r = 1
    return t
end function
```

A.2 Polynomial Transforms

The inverse and forward transforms are almost identical, except the outer loop direction and the inner operations are reversed.

Notice that the transforms can use factors of a greater power than needed. To compute multiple transforms of different sizes with the same offset, only the factors for the largest size must be computed, and can be used for all smaller sizes. This is very convenient for the error

Algorithm 6 Transform Algorithms

function INVERSETRANSFORM(data, factors)
  for step $\leftarrow 0$ **to** $\log_2(\text{len(data)}) - 1$ **do**
    group_len $\leftarrow 2^{\text{step}}$
    group_factors_start $\leftarrow$ len(factors) $+ 1 - \frac{\text{len(factors)}+1}{2^{\text{step}}}$
    for group $\leftarrow 0$ **to** $\frac{\text{len(data)}}{2^{\text{step}+1}} - 1$ **do**
      for x $\leftarrow 0$ **to** group_len $- 1$ **do**
        $a \leftarrow$ group $\cdot$ group_len $\cdot\, 2 + x$
        $b \leftarrow a +$ group_len
        data$[b] \leftarrow$ data$[b] +$ data$[a]$
        data$[a] \leftarrow$ data$[a] +$ data$[b] \cdot$ factors[group_factors_start + group]
      end for
    end for
  end for
end function

function FORWARDTRANSFORM(data, factors)
  for step $\leftarrow \log_2(\text{len(data)}) - 1$ **down to** 0 **do**
    group_len $\leftarrow 2^{\text{step}}$
    group_factors_start $\leftarrow$ len(factors) $+ 1 - \frac{\text{len(factors)}+1}{2^{\text{step}}}$
    for group $\leftarrow 0$ **to** $\frac{\text{len(data)}}{2^{\text{step}+1}} - 1$ **do**
      for x $\leftarrow 0$ **to** group_len $- 1$ **do**
        $a \leftarrow$ group $\cdot$ group_len $\cdot\, 2 + x$
        $b \leftarrow a +$ group_len
        data$[a] \leftarrow$ data$[a] +$ data$[b] \cdot$ factors[group_factors_start + group]
        data$[b] \leftarrow$ data$[b] +$ data$[a]$
      end for
    end for
  end for
end function

function PRECOMPUTEFACTORS(pow, offset)
  factors $\leftarrow$ new array of GF(2^{64}) values of size $2^{\text{pow}} - 1$
  factor_idx $\leftarrow 0$
  for step $\leftarrow 0$ **to** pow $- 1$ **do**
    groups $\leftarrow 2^{\text{pow}-\text{step}-1}$
    for group $\leftarrow 0$ to groups $- 1$ **do**
      factors[factor_idx] $\leftarrow \hat{W}_{\text{step}}(\omega_{\text{group}\cdot 2^{\text{step}+1}} + \omega_{\text{offset}})$
      factor_idx $\leftarrow$ factor_idx $+ 1$
    end for
  end for
  return factors
end function

locator polynomial computation algorithm, which uses transforms of many different sizes.

The precomputed factors require $O(n \log n)$ time and space complexity.

A.3 Formal Derivative

The formal derivative also has time complexity $O(n \log n)$, but unlike the transforms, the factors only require $O(\log n)$ space.

Algorithm 7 Formal Derivative Algorithm

function PRECOMPUTEDERIVATIVEFACTORS(pow)
    assert $0 \leq$ pow ≤ 64
    factors $\leftarrow$ new array of GF(2^{64}) values of size pow
    for $l \leftarrow 1$ **to** pow $- 1$ **do**
        for $j \leftarrow 2^{l-1}$ **to** $2^l - 1$ **do**
            factors$[l] \leftarrow$ factors$[l] * \omega_j$
        end for
        if $l + 1 \neq$ pow **then**
            factors$[l + 1] \leftarrow$ factors$[l]$
        end if
        factors$[l] \leftarrow$ factors$[l]/W_l(2^l)$
    end for
    return factors
end function
function FORMALDERIVATIVE(data, factors)
    for $i \leftarrow 0$ **to** len(data) - 1 **do**
        for bit $\leftarrow 0$ **to** $\log_2$(len(data)) **do**
            if $i \;\&\; 2^{\text{bit}} \neq 0$ **then**
                data$[i - 2^{\text{bit}}] \leftarrow$ data$[i - 2^{\text{bit}}]$ + data$[i] \cdot$ factors[bit]
            end if
        end for
        data$[i] \leftarrow 0$
    end for
end function

A.4 Error Locator Polynomial

The algorithm splits the error locator polynomial at each step, recursively computing the coefficients for each half. The halves are multiplied together by padding with zeros, converting to values, multiplying, then converting back to coefficients.

The complexity is $O(n \log^2 n)$, since each step combines the results of two smaller steps in $O(n \log n)$ time using the transforms.

Algorithm 8 Error Locator Polynomial Computation

function COMPUTEERRORLOCATOR(erasures, out_len, t_fac, d_fac)
  values ← new empty array
  coefficients ← InternalRecursion(erasures, out_len, t_fac, d_fac, values)
  FormalDerivative(coefficients, d_fac)
  ForwardTransform(coefficients, t_fac)
  return (values, coefficients) ▷ coefficients now contains values of derivative
end function
function INTERNALRECURSION(erasures, out_len, t_fac, out_values)
  if len(erasures) $= 1$ **then**
    if out_values $\neq$ null **then**
      out_values ← new array $[\omega_i + \omega_{\text{erasures}[0]}$ **for** i **from** 0 **to** out_len $- 1]$
    end if
    return new array $[\omega_{\text{erasures}[0]}, 1, 0, \ldots, 0]$ of size out_len
  end if
  special_case ← len(erasures) + 1 = out_len
  $a \leftarrow$ InternalRecursion(erasures **from** 0 **to** $\frac{\text{len(erasures)}}{2} - 1$, $\frac{\text{out\_len}}{2}$, t_fac, null)
  ResizeWithZeros(a, out_len)
  ForwardTransform(a, t_fac)
  $b \leftarrow$ InternalRecursion(erasures **from** $\frac{\text{len(erasures)}}{2}$ + special_case **to end**, $\frac{\text{out\_len}}{2}$, t_fac, null)
  ResizeWithZeros(b, out_len)
  ForwardTransform(b, t_fac)
  $a \leftarrow a * b$ ▷ polynomial evaluations are multiplied in $O(n)$ time
  if special_case **then**
    $a \leftarrow a * [\omega_i + \omega_{\text{erasures}[\frac{\text{len(erasures)}}{2}]}$ **for** i **from** 0 **to** $\frac{\text{out\_len}}{2} - 1]$ ▷ multiply in extra value
  end if
  if out_values $\neq$ null **then** ▷ the top-most call must return both coefficients and values
    out_values ← Copy(a) ▷ the memory of b can be reused here for the copy
  end if
  InverseTransform(a, t_fac) ▷ convert back to coefficients after multiplications are done
  return a
end function

The special case is sometimes needed to prevent a branch where len(erasures) $=$ out_len, which would request only n coefficients for a polynomial of degree n.

A.5 Recovery Figures

To visually demonstrate some limitations of this error correction scheme, the following figures show the use of Reed-Solomon codes to repair bitmap images.

Figure (a) has 12966 errors and can be repaired, yet figure (b) has only 3288 errors and cannot be repaired. This is because the errors in figure (a) are one contiguous burst, whereas the errors in figure (b) are many small bursts spread across the image, damaging more blocks than in figure (a).

(a) A recoverable bitmap image with one corrupt area.

(b) An unrecoverable bitmap image with many small corrupt areas.

(c) Figure (a), repaired.

Figure 1: Image source: `scipy.datasets.face` (derived from https://pixnio.com/fauna-animals/raccoons/raccoon-procyon-lotor)

A.5.1 Image Generation Script

```
from os import system
from scipy.datasets import face as get_face
from imageio.v3 import imwrite
from PIL import Image
import numpy as np

def main():
    face = get_face(gray=True)
    face = Image.fromarray(face).resize((256, 256))
    imwrite("face.bmp", np.array(face))
    system("cargo run encode face.bmp face.rsarc 4096 5")

    rng = np.random.default_rng(42)
    def randomize(arr):
        for i in range(0, len(arr)):
            arr[i] = rng.integers(0, 256)

    face = np.array(face)
    face_2 = face.copy()
    face_3 = face.copy()
    randomize(face_2.ravel()[7300:20300])

    for i in range(0, face_3.size, 2000):
        randomize(face_3.ravel()[i:i+100])

    print(f"Differences between face and face_2: {np.sum(face != face_2)}")
    print(f"Differences between face and face_3: {np.sum(face != face_3)}")

    imwrite("face_2.bmp", face_2)
    imwrite("face_2_repaired.bmp", face_2)
    imwrite("face_3.bmp", face_3)
    imwrite("face_3_repaired.bmp", face_3)

    system("cargo run repair face_2_repaired.bmp face.rsarc")
    system("cargo run repair face_3_repaired.bmp face.rsarc")

    face_2_repaired = Image.open("face_2_repaired.bmp")
    face_3_repaired = Image.open("face_3_repaired.bmp")
    assert np.all(np.array(face_2_repaired) == face)
    assert not np.all(np.array(face_3_repaired) == face)

    def to_png(path):
        imwrite(f"{path}.png", np.array(Image.open(f"{path}.bmp")))

    to_png("face_2")
    to_png("face_3")
    to_png("face_2_repaired")

if __name__ == "__main__":
    main()
```

A.6 Dependencies

The following Rust libraries were used in the implementation:

- `fastrand` - Random number generation for testing.
- `blake3` - Hashing for error detection.
- `crossbeam-channel` - Channels for communication between threads. Although the standard library provides channels, multi-producer multi-consumer channels are not stabilized yet (as of Rust 1.87.0).
- `indicatif` - Terminal progress bars.
- `memmap2` - Cross-platform memory mapped I/O.
- `num_cpus` - Obtains of CPU cores for automatically selecting number of threads.
- `positioned-io` - Cross-platform random access file I/O.
- `sysinfo` - Obtains available memory for automatically selecting number of codes to process at once.

Bibliography

[1] Shay Gueron and Michael E. Kounavis. *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. 2014. URL: https://web.archive.org/web/20190806061845/https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf (visited on 2025-01-31).

[2] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. 1997. URL: https://www.cambridge.org/core/books/finite-fields/75BDAA74ABAE713196E718392B9E5E72 (visited on 2025-01-31).

[3] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. "Novel Polynomial Basis and Its Application to Reed-Solomon Erasure Codes". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. 2014, pp. 316–325. DOI: 10.1109/FOCS.2014.41. URL: https://arxiv.org/pdf/1404.3458 (visited on 2025-05-28).

[4] F. Jessie MacWilliams and N. J. A. Sloane. "The Theory of Error-Correcting Codes". In: 1977. URL: https://api.semanticscholar.org/CorpusID:118260868 (visited on 2025-01-31).

[5] Gadiel Seroussi. *Table of Low-Weight Binary Irreducible Polynomials*. 1998. URL: https://shiftleft.com/mirrors/www.hpl.hp.com/techreports/98/HPL-98-135.pdf (visited on 2025-01-31).

(Acest document se leagă împreună cu lucrarea de licență)

Declarație

Subsemnatul/Subsemnata NEGRESCU THEODOR, candidat la examenul de licență, sesiunea IUNIE-IULIE 2025, la Facultatea de Matematică şi Informatică, programul de studii Informatică, declar pe propria răspundere că lucrarea de faţă este rezultatul muncii mele, pe baza cercetărilor mele și pe baza informaţiilor obţinute din surse care au fost citate şi indicate, conform normelor etice, în note și în bibliografie. Declar că nu am folosit în mod tacit sau ilegal munca altora și că nici o parte din teză nu încalcă drepturile de proprietate intelectuală ale altcuiva, persoană fizică sau juridică. Declar că lucrarea nu a mai fost prezentată sub această formă vreunei instituţii de învăţământ superior în vederea obţinerii unui grad sau titlu ştiinţific ori didactic.

Data	Semnătura
01.06.2025