# PyRS: Cross-platform Data Fault-tolerant Storage Library Based on RS Erasure Code

Junqiang Ma[1], Weihao Yan[2], Xiaotian Zhang[2], Min Huang[2*], Jingyang Wang[2]

[1] Department of Electrical Engineering, Hebei Vocational University of Technology and Engineering, China
[2] School of Information Science and Engineering, Hebei University of Science and Technology, China
22538805@qq.com, 1793131106@qq.com, 624943523@qq.com, huangmin@hebust.edu.cn, jingyangw@hebust.edu.cn

## Abstract

Erasure code has been used by more and more researchers to solve the problem of efficient, reliable, and fault-tolerant data storage. However, the existing libraries based on erase code can only run on the Linux platform, and some of them need GPU support. This paper implements a cross-platform data fault-tolerant storage library based on RS erasure code, PyRS, which is running on CPU without GPU support and developed in Python. PyRS uses Vandermonde matrix as the coding matrix and Numba and NumPy libraries to speed up and optimize the program. This paper compares PyRS with Jerasure on the same Linux platform. The results show that the encoding and decoding speed of PyRS is 8 times faster than that of Jerasure. The CPU usage rate of both is about 25% and the memory usage rate of PyRS is about 5% higher than that of Jerasure. The same experiments are carried out on PyRS on the Windows platform. Experimental results show that compared with running on the Linux platform, PyRS running on the Windows platform has almost the same speed of encoding and decoding, and its CPU usage rate increases by about 15%, while its memory usage rate decreases by about 5%.

**Keywords:** Data fault-tolerant storage, RS erasure code, Jerasure, Vandermonde matrix, Cross-platform

## 1 Introduction

Erasure code is a forward error correction technology that can ensure data reliability with more optimized data redundancy [1]. Erasure code was first used to solve the problem of data transmission loss in the field of communication, which introduced erasure code technology into the storage field due to its advantages of preventing data loss.

Today, erasure code technology has been widely used in the storage field and has become one of the leading fault-tolerant technologies in storage systems [2]. Another major fault-tolerant technology for storage systems is replication [3]. Compared with the traditional replication technology, erasure code technology has the advantages of low redundancy and high reliability. For example, when we store a file of size 10KB with replication technology of the 3-copies policy, the storage cost is 30KB, because the same data is stored in three copies. However, erasure code technology is used to realize the storage effect of the 3-copies policy, on the premise that the size of each data block is 1KB, the file is divided into 10 segmented data blocks and 3 redundancy check data blocks are generated, and the storage cost is 13KB. Therefore, erasure code technology can ensure data reliability with lower storage cost than replication technology.

Both Hadoop3.0 and Ceph distributed storage systems use erasure code technology as one of the system fault-tolerance technologies. At the same time, Microsoft, Google, Facebook, Amazon, Taobao, and other Internet giants have also begun to study erasure code storage technology and apply it to their respective mainstream storage systems, such as Microsoft's cloud storage system Azure [4] and Facebook's warm BLOB storage system [5].

### 1.1 Literature Review

At present, with the scale of data storage increasing, the data storage space is facing severe challenges, and the erasure code technology can solve this problem well. Therefore, a flexible and convenient erasure code library is vital for the storage system.

In 2007, Jerasure was published, which was an erasure code library and developed by James S. Plank [6]. The Jerasure library had complete functions and supported a variety of erasure codes, including RS (Reed-Solomon) coding and CRS coding. However, it is developed in the C language and has platform limitations. Jerasure can only support the Linux system, and its speed performance is lower. In 2015, Chu et al. designed and implemented the PErasure library [7], which was a parallel Cauchy Reed-Solomon (CRS) EC library, used Graphics Processing Units (GPUs) to complete the EC computing task and improved the speed of the EC library to some extent. However, it cannot fully use the GPU memory system, resulting in poor performance. In 2016, Dai et al. used CUDA to accelerate RS code based on the reed_sol_r6_OP (R6) algorithm of Jerasure library [8] and achieved good results, with speed up to 20 times that of the original R6 algorithm. But it only supports Linux systems.

In 2017, Zhou proposed an improved coding based on RS erasure code – LRC [9]. Compared with RS, LRC could save nearly half of the decoding cost in the case of similar storage costs. When the encoding parameters were changed, the codec performance of LRC did not change significantly, and more parameter combinations could be selected. However, the tool

*Corresponding Author: Min Huang; E-mail: huangmin@hebust.edu.cn
DOI: 10.53106/160792642022122307014

is developed in C language and can only run on Linux. Gong et al. designed a new erasure code, ZD code [10]. Two structures based on Vandermonde matrix and Hankel matrix were proposed. The codec performance of ZD code and Cauchy-RS code was compared. The decoding performance of ZD code was at least 20% better than that of Cauchy-RS code. However, the tool is implemented in C language, which has the limitation of cross-platform.

In 2018, Liu et al. designed and implemented G-CRS [11], a new GPU-based CRS erasure code library. Compared with PErasure, this library can make full use of GPU resources to improve the EC's performance to some extent. Liu et al. proposed an efficient single-disk fault coding scheme, L-Code [12], which used both horizontal parity check and anti-angle parity check for data reconstruction. The scheme improved the performance of single-fault disk reconstruction. But L-Code can only run on Linux.

In 2019, Xie et al. proposed an Availability Zone erasure Code (AZ-Code) [13], which took advantages of MSR and LRC codes. Experiments showed that AZ-Code could keep low recovery costs and high reliability compared with traditional erasure code. But it only runs on the Hadoop and Linux system. In 2020, Qiu et al. proposed an efficient hybrid erasure encoding framework EC-Fusion in Cloud Storage Systems [14], which combined RS and MSR code. Compared with traditional erasure code, EC-Fusion improved application response time and reduced reconstruction time. But experiments are only taken on the Hadoop platform under the Ubuntu system.

In 2021, Fang et al. proposed a new CLRC algorithm based on RS algorithm [15], which generated local check blocks by grouping RS encoded blocks. Compared with RS algorithm, CLRC reduced the bandwidth and I/O consumption of data recovery. However, the algorithm is tested on the Centos platform, and there are limitations in cross-platform. Arslan proposed an erasure code library - Founsure [16], which could save storage space, minimize data storage overhead, and reduce data recovery bandwidth. But Founsure is implemented by Linux instructions.

To sum up, all research achievements can only run on Linux platforms, do not have the cross-platform capability and cannot run on Windows platform. Some achievements require Hadoop environment support. The research achievements in reference [7, 11] also rely on GPUs, which undoubtedly increases the cost of using the system. Therefore, it is of great significance to design and develop a cross-platform data fault-tolerant storage library based on RS erasure code that can run on CPU.

## 1.2 Main Contributions

PyRS is developed in Python. It takes advantage of the cross-platform and flexibility of the Python. PyRS contributes to the universal application of erasure code technology in a storage system. The contributions of this paper are as follows:

- PyRS is a cross-platform data fault-tolerant storage Library, which can run on Linux and Windows platforms. It is easy to operate PyRS, and PyRS's running environment is easily configured.
- PyRS uses Numba and NumPy to improve its performance in terms of speed. Numba can compile some of the programs into local machine instructions, and NumPy has high performance for processing data.
- PyRS can be up to 8 times faster than the Jerasure library in the same system environment and state.
- PyRS library has the advantages of fast speed, cross-platform, simple operation, and accessible environment configuration.

## 1.3 Organization

The remaining parts of this paper are organized as follows: Section II introduces the encoding and decoding principles of RS erasure code. In Section III, we first introduce the Galois field and encoding matrix, and then describe the implementation process of PyRS data encoding and decoding, finally, explain the process of Numba and NumPy accelerating PyRS. In Section IV, we compare PyRS with Jerasure on Linux Platform in terms of encoding/decoding speed, CPU usage and memory usage, and then test PyRS on the Windows Platform. The last section summarizes the work of this paper and introduces future work.

# 2 RS Erasure Code

RS erasure code is a multi-array BCH code with robust error correction ability, dividing the data file into k segmented data blocks in bytes and then generating m redundancy check data blocks with equal size through encoding. When restoring the file data, within the loss of m data blocks, any k data blocks (including segmented data blocks and redundancy check data blocks) selected from the remaining data blocks can be decoded to obtain the original data file [17].

## 2.1 Principle of RS Erasure Code Encoding

The progress of RS erasure code coding is as follows:

(1) The data file is divided into $D_1, D_2, D_3, \ldots, D_k$ , total k segmented data blocks in bytes.

(2) $D = (D_1, D_2, D_3, \ldots, D_k)$ as input vector, V is an encoding matrix.

(3) D multiplies with the encoding matrix to get the result data Y.

$$V * D = Y. \tag{1}$$

$$V = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ B_{11} & B_{12} & \cdots & B_{1k} \\ B_{21} & B_{22} & \cdots & B_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mk} \end{bmatrix}. \tag{2}$$

As shown in formulas (1) and (2), V is an encoding matrix, and any k*k sub-matrix must be invertible, as shown in formula (3). $D = (D_1, D_2, D_3, \ldots, D_k)$ is the input vector, $Y = (D_1, D_2, D_3, \ldots, D_k, C_1, C_2, \ldots, C_m)$ is the encoding result, as shown in formula (4), and formula (4) is equivalent to formula (1).

$$B = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1k} \\ B_{21} & B_{22} & & B_{2k} \\ \vdots & & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mk} \end{bmatrix}. \quad (3)$$

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ B_{11} & B_{12} & \cdots & B_{1k} \\ B_{21} & B_{22} & \cdots & B_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mk} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_k \\ C_1 \\ C_2 \\ \vdots \\ C_m \end{bmatrix}. \quad (4)$$

As shown in formulas (2) and (3), it is easy to find that the matrix V consists of the identity matrix and the matrix B. V*D is equivalent to first multiplying the identity matrix with the input vector D, and then multiplying the matrix B with the input vector. The identity matrix is multiplied by the input vector D, and the result is obviously the input vector, where the elements represent k equally-sized segmented data blocks. The matrix B is multiplied by the input vector, and the result is redundancy check data blocks. The vertical combination of the two results is the encoding result. Therefore, the essence of RS erasure code coding is to divide the data file into k segmented data blocks of equal size, then multiply the matrix B with the input vector to obtain redundancy check data blocks [18]. To ensure that matrix V's k*k sub-matrix is invertible, matrix B is generally a Vandermonde matrix or a Cauchy matrix.

## 2.2 Principle of RS Erasure Code Decoding

RS erasure code decoding refers to recovering the original data file using encoded data blocks generated by RS erasure code encoding and mathematical algorithms [19-20]. The process is as follows.

(1) Assuming that the encoded data blocks $D_1$ and $C_1$ have been lost, the rows corresponding to the missing data blocks are deleted from the encoding matrix. That is, the first row and the (k+1)th row of matrix V are deleted, and the new matrix after deletion is denoted as $V'$.

$$V'' = \begin{bmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ B_{21} & B_{22} & \cdots & B_{2k} \end{bmatrix}. \quad (5)$$

$$Y'' = \begin{bmatrix} D_2 \\ \vdots \\ D_k \\ C_2 \end{bmatrix}. \quad (6)$$

(2) Take any k-row vector from the matrix $V'$ to get the matrix $V''$. As shown in formulas (5) and (6), we select k-row vectors from top to bottom, so the matrix $V''$ is a k*k square matrix, and the corresponding vector of the encoded data block is $Y''$.

(3) According to formulas (4), (5), and (6), the following formula(7) can be obtained.

$$V'' * D = Y''. \quad (7)$$

(4) The inverse matrix of V" multiplied by both sides of the formulas (7), then the formulas (8) and (9) can be obtained.

$$(V'')^{-1} * V'' * D = (V'')^{-1} * Y''. \quad (8)$$

$$D = (V'')^{-1} * Y''. \quad (9)$$

(5) At this point, the original data block D is obtained.

# 3 PyRS Implementation

We use flexible and cross-platform Python to develop PyRS. Vandermonde matrix is selected as the encoding matrix. Any sub-matrix of the Vandermonde matrix is a singular matrix with an inverse matrix. The Gaussian elimination method calculates the inverse matrix in RS encoding/decoding. Four arithmetic operations defined in Galois field $GF(2^w)$are used to solve the problem of addition, subtraction, multiplication, and division of real numbers. The PyRS system model is shown in Figure 1.

(1) Galois field. Galois field $GF(2^w)$ is constructed by building the $2^w$ distinct elements on the Galois field. The finite field $GF(2^w)$ is constructed using an irreducible polynomial of $GF(2^w)$, and the four arithmetic operations on the Galois field are realized simultaneously. The results of the four arithmetic operations on the Galois field will fall into the finite field, which avoids the problem of data overflow caused by too large numerical results of the mathematical calculation and ensures the accuracy of the data.

(2) Encoding matrix. The encoding matrix is constructed using a Vandermonde matrix to meet the needs of RS erasure code encoding/decoding.

(3) Encoding. The data encoding based on RS erasure code can divide any file into k segmented data blocks and simultaneously generate m redundancy check data blocks. The result of encoding is k+m data blocks, including k segmented data blocks and m redundancy check data blocks. When no more than m data blocks fail at the same time, the original file can still be restored by remaining any k non-failed data blocks, thereby, the availability of the storage system is ensured, and the reliability is improved [21-22].

(4) Decoding. It is assumed that the file is divided into k segmented data blocks, $D_1, D_2, D_3, \dots, D_k$. Simultaneously use RS erasure code to generate m redundancy check data blocks, $C_1, C_2, \dots, C_m$. The size of all blocks is both t bytes. When no more than m data blocks in $D_1, D_2, D_3, \dots, D_k, C_1, C_2, \dots, C_m$ are invalid, PyRS can perform decoding and reconstruction calculation through any remaining k data blocks that have not failed to restore the original file data.

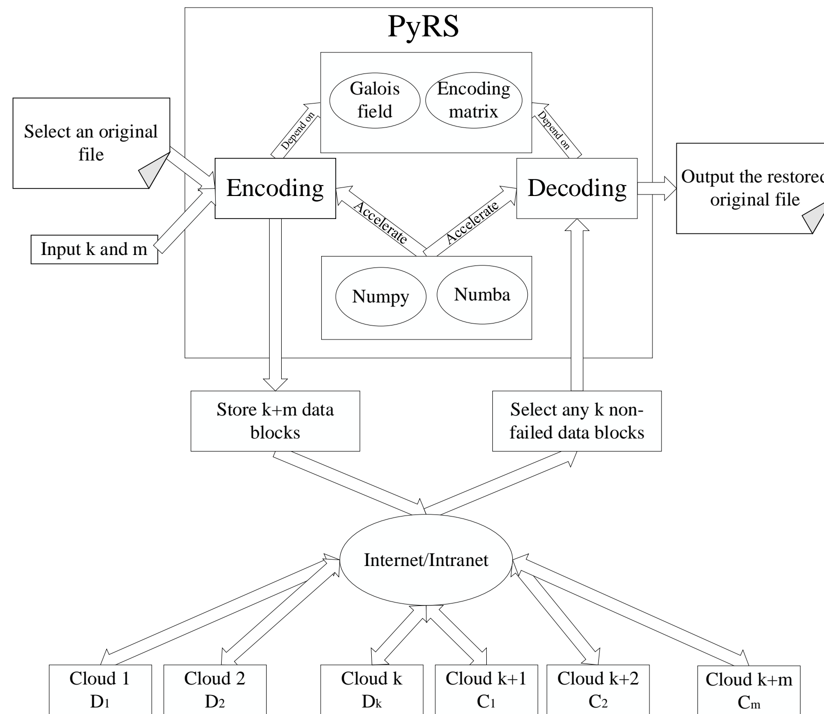



**Figure 1.** PyRS system model

(5) Numba and NumPy technologies accelerate PyRS. Python has the advantages of high development efficiency and cross-platform, so the coding library introduced in this paper is implemented in Python. However, the execution efficiency of the program written in Python is lower, so the execution speed of PyRS needs to be improved. Aiming at the characteristic of PyRS using many mathematical calculations and loop statements, this paper selects the Numba library to accelerate. The Numba library can compile the corresponding Python program into local machine instructions at runtime to achieve the purpose of acceleration. Besides, the high-performance characteristics of NumPy are utilized to process data and improve the speed performance of PyRS.

To ensure the safe and fault-tolerant storage of file data, more and more users not only need to divide file data into multiple data blocks and store them on different disks in a distributed storage system or different clouds in multi-cloud storage, but also need to store the same file data block repeatedly through replication technology. Although the implementation of replication technology is simple, the storage cost is very high. Erasure code technology can realize efficient storage. The process of safe and fault-tolerant storage of file data through PyRS is as follows:

(1) Select an original file. Select a file of any format type on your local disk. This file is to be stored securely.

(2) Input k and m. k indicates the number of blocks to divide the original file into data blocks. m represents the number of redundant data blocks generated by PyRS.

(3) Encoding. The data encoding of PyRS divides the original file into k segmented data blocks and simultaneously generates m redundancy check data blocks. There are k+m data blocks.

(4) Store k+m data blocks. k+m data blocks are stored in different clouds of a multi-cloud storage through Internet/Intranet. We can also store the data on different disks in a distributed storage system.

The data of k+m data blocks stored in a multi-cloud storage or a distributed system may be invalid due to disk damage, viruses or hacker attacks, and network faults. As long as the data blocks stored by PyRS fail less than m data blocks, the original file data can be correctly restored through PyRS decoding. Thus, the fault-tolerant storage of file data is realized. The process for PyRS to restore file data is as follows:

(1) Select any k non-failed data blocks. Obtain any k non-failed data blocks from k + m data blocks stored in the multi-cloud storage systems or distributed systems through Internet/Intranet.

(2) Decoding. The data decoding of PyRS restores the original file using the selected k non-failed data blocks.

(3) Output the restored original file.

## 3.1 Galois Field

### 3.1.1 Galois Field Structure

As a finite field, Galois field can reasonably ensure that the results of four arithmetic operations fall within the finite field in the encoding/decoding process. The accuracy of the data and the successful implementation of file encoding and recovery are ensured. The number of bits in a byte is 8, and its value range is 0~255. Therefore, w=8, that is, $GF(2^8)$ is selected in this paper to define and construct the Galois field, and the elements in the field are represented using an 8-bit binary integer [23].

The polynomial in $GF(2^8)$ is as formula (10).

$$f(x) = a_7x^7 + a_6x^6 + \cdots + a_1x + a_0. \tag{10}$$

In formula (10), the element values in $(a_0, a_1, \ldots, a_7)$ can only take 0 or 1, so $GF(2^8)$ has $2^8 = 256$ different polynomials, each polynomial of $GF(2^8)$ can be expressed as an 8-bit binary integer. It can be seen that the finite field $GF(2^8)$ can be composed and represented by the coefficients

of its polynomial, that is, the binary representation. The decimal set corresponding to binary is the decimal representation of the finite field $GF(2^8)$, which is also the elemental representation of the Galois field.

This paper uses the method of a generator to define and construct the Galois field, and the four arithmetic operations of the Galois field are completed by using the table lookup method. The generator g is an element in the finite field $GF(2^w)$. The first 2w-1 power of g can constitute all non-zero elements of the finite field $GF(2^w)$. The elements of the field $GF(2^w)$ are $g^0, \dots, g^{w-2}$. The generator g can generally be taken as the root of an irreducible polynomial in the field $GF(2^w)$. In this paper, the irreducible polynomial of the finite field $GF(2^w)$ is shown in formula (11). If $f(g) = 0$, formula (12) can be obtained. Therefore, the elements in the finite field $GF(2^w)$ can be expressed as: $0, 1, g^0, \dots, g^7, g^8 = g^4 + g^3 + g^2 + 1, g^9 = g * g^8 = g^5 + g^4 + g^3 + g, g^{10} = g * g^9 = g^6 + g^5 + g^4 + g^2, \dots, g^0$.

$$f(x) = x^8 + x^4 + x^3 + x^2 + 1. \quad \textbf{(11)}$$

$$g^8 = g^4 + g^3 + g^2 + 1. \quad \textbf{(12)}$$

The generator, polynomial, binary, and decimal representation of Galois field $GF(2^8)$ with the formula (11) as the irreducible polynomial are shown in Table 1.

**Table 1.** $GF(2^8)$ generator, polynomial, binary, decimal representation

| Generator | Polynomial | Binary | Decimal |
|---|---|---|---|
| 0 | 0 | 00000000 | 0 |
| $g^0$ | $g^0$ | 00000001 | 1 |
| $g^1$ | $g^1$ | 00000010 | 2 |
| $g^2$ | $g^2$ | 00000100 | 4 |
| $g^3$ | $g^3$ | 00001000 | 8 |
| $g^4$ | $g^4$ | 00010000 | 16 |
| $g^5$ | $g^5$ | 00100000 | 32 |
| $g^6$ | $g^6$ | 01000000 | 64 |
| $g^7$ | $g^7$ | 10000000 | 128 |
| $g^8$ | $g^4+g^3+g^2+1$ | 00011101 | 29 |
| $g^9$ | $g^5+g^4+g^3+g$ | 00111010 | 58 |
| $g^{10}$ | $g^6+g^5+g^4+g^2$ | 01110100 | 116 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $g^{255}$ | $g^0$ | 00000001 | 1 |

According to the characteristics of the Galois field constructed by using the generator method, this paper builds three tables, namely positive table, anti-table, and inverse table.

The positive table stores the elements of the finite field $GF(2^8)$. The table index is the exponent of the generator $g$, and the value range is [0, 254]. The element in the table with index k represents $g^k$, and its value range [1, 255].

The anti-table stores all the exponents of the generator of the finite field $GF(2^8)$, which index is the decimal value of polynomial $g^k$, and the value range is [1, 255].

The element corresponding to the index is the exponent of the generator, and the value range is [0, 254].

In the inverse table, the index value and the value of the corresponding element of the index are mutually inverse, and the value range is [1, 255].

$GF(2^w)$ is a finite field; that is, the number of elements in the field is limited, but the exponent k of the generator is infinite. So, there must be cycles. The period of this cycle is 2w-1, because g cannot generate polynomial 0, so for $GF(2^8)$, when k is greater than or equal to 255, $g^k = g^{(k\%255)}$. Therefore, for the positive table, the exponent of the generator element can be 0 to 254, and correspondingly 255 different polynomials are generated, and the value range of the polynomial is 1 to 255.

(1) Construct the main table. Define the main table named gf_table, let gf_table[255] = 0 and gf_table[0] = 1. According to $g^k = g * g^{k-1}$, $(k \in [0,254])$, the value of the corresponding element of the higher exponent of the generator is equal to the result of shifting the value of the corresponding element of the next exponent to left by 1 bit, and the premise is that the corresponding polynomial exponent is less than 8. When the element is shifted left 1 bit, the corresponding polynomial exponent equals 8, and it needs to be processed according to formula (12). That is, replace $x^8$ with $x^4 + x^3 + x^2 + 1$, and then perform XOR operation to get element value.

(2) Construct the anti-table. Define the anti-table named gf_arc_table, and let gf_arc_table[0] = 0. The index of gf_arc_table represents the polynomial value, and the elements in the table represent the exponent of the generator. Take the element value in the main table, use it as the index of the anti-table, and then use the corresponding index of the element value in the main table as the value of the anti-table element.

(3) Construct the inverse table. Define the inverse table gf_inverser_table, and let the index value and the value of the corresponding element of the index be inverses of each other. For example, the value of an element in the inverse table $v_1$, the corresponding inverse table index is i. Using the main table and anti-table defined in (1) and (2), take the element value $v_2$ at index i from the anti-table. Let the maximum value gf_maxnum in the Galois field defined in this paper minus $v_2$; we can get the exponent of the generator corresponding to the inverse of $v_1$ in the Galois field $GF(2^8)$. Taking this exponent as the index, the element value obtained through the main table is $v_1$.

### 3.1.2. Four Arithmetic Operations in Galois Field

The four arithmetic operations in the Galois field are as follows.

(1) The addition of Galois field $GF(2^8)$. That is, the result of the addition is XOR the two input values.

(2) The subtraction of Galois field $GF(2^8)$. The same as the addition.

(3) The multiplication of Galois field $GF(2^8)$. $GF(2^8)$ multiplication is the multiplication of two elements x and y in the field, i.e., x*y. First, obtain the exponents $k_1$ and $k_2$ of the generator g corresponding to the two elements (x and y) by looking up the table gf_arc_table. And then, according to the characteristics of exponent multiplication, i.e., $x * y = g^{k_1} * g^{k_2} = g^{(k_1+k_2)} = g^k$, we can get the representation of generator $g^k$. Finally, we obtain the decimal representation corresponding to $g^k$ by looking up the table gf_table. When x is 0 or y is 0, x*y cannot be calculated through the main table and anti-table. Therefore, the multiplication implementation adds the judgment of whether the multiplier is 0. If the

multiplier is 0, the result is 0; otherwise, the result is calculated by looking up the table. The pseudo-code for $GF(2^8)$ multiplication is shown in Algorithm 1.

**Algorithm 1.** The pseudo-code For $GF(2^8)$ multiplication

1.Enter two numbers:$x, y$;
2.**if** $x == 0$ or $y == 0$ **then**
3. **return** 0;
4.**end if**
5. $res1 \leftarrow gf_arc_table[x] + gf_arc_table[y]$;
6.**if** $res1 < gf_maxnum$ **then**
7. **return** $gf_table[res1]$;
8.**else**
9. **return** $gf_table[res1\%gf_maxnum]$;
10.**end if**

(4) The division of Galois field $GF(2^8)$. $GF(2^8)$ division refers to dividing two elements x and y in the field, i.e., x/y. x divided by y is equivalent to x multiplies the inverse of y. So, we can think of division as multiplication. Therefore, the inverse of y can be obtained by checking the inverse table gf_inverser_table, and then x and the inverse of y are multiplied. The pseudo-code for $GF(2^8)$ division is shown in Algorithm 2.

**Algorithm 2.** The pseudo-code for $GF(2^8)$ division

1.Enter two numbers: $x, y$;
2.**if** $y == 0$ **then**
3. Throw an exception and end the program;
4.**end if**
5. $y^{-1} \leftarrow gf_inverser_table[y]$;
6.Mutiply $x$ and $y^{-1}$ ;

## 3.2 Encoding Matrix

This paper selects the Vandermonde matrix to generate the encoding matrix based on the Galois field algorithm. The Vandermonde matrix is shown in formula (13).

$$Van = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & k \\ 1^2 & 2^2 & 3^2 & \cdots & k^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1^{m-1} & 2^{m-1} & 3^{m-1} & \cdots & k^{m-1} \end{bmatrix}. \quad (13)$$

Construct the identity matrix and combine it with the Vandermonde matrix to get the encoding matrix, as shown in formula (14).

$$V = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & k \\ \vdots & \vdots & \ddots & \vdots \\ 1^{m-1} & 2^{m-1} & \cdots & k^{m-1} \end{bmatrix}. \quad (14)$$

## 3.3 Implementation Process of Data Encoding Based on RS Erasure Code

The pseudo-code of the encoding process is shown in Algorithm 3.

**Algorithm 3.** Pseudo-code of the encoding process

1. Enter encoding information
2. Initialize file read-write buffer
3. Construct Galois field
4. Construct coding matrix
5. *loop:*
6. Read file
7. **if** Read content size is smaller than the size of read buffer **then**
8. Data block zero padding
9. **end if**
10. Split file data into blocks
11.Generate check matrix
12.**if** First encoding **then**
13. Write file
14.**else**
15. Append file
16.**end if**
17.**if** Last encoding **then**
18. Write encoding information file
19.**else**
20. **goto** *loop*.
21.**end if**

Figure 2 shows the implementation process of data encoding.

The main steps to achieve encoding are as follows:

(1) Enter the encoding information. Enter the name of the file to be encoded, the number k of equally divided blocks, the number m of redundancy check blocks, the buffer size for reading file, etc.

(2) Initialize the buffer size for reading file. Refer to the file size, initialize the buffer size so that the file data can be divisible by the input number of equally divided blocks k. The file data to be encoded can be divided into k blocks of equal size.

(3) Construct Galois field. Construct the finite field $GF(2^8)$ and define four arithmetic operations on the finite field as the basis and premise of all operations in encoding and decoding.

(4) Construct an encoding matrix.

(5) Encoding. Use the file reading buffer obtained in step (2) to read the file data multiple times. Each time the file is read, it needs to be encoded once. The number of times file readings is the number of times of encodings.

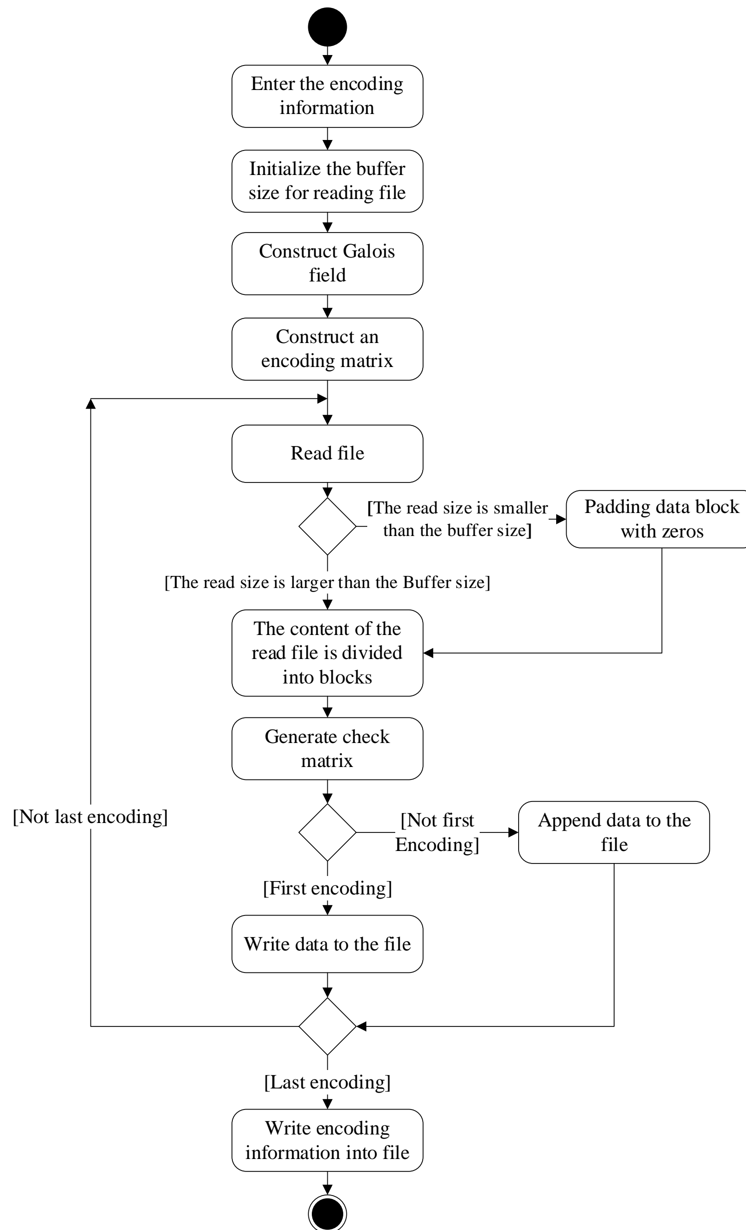



**Figure 2.** Activity diagram of data encoding process

The main steps are as follows:

1) Read file. The file will be read multiple times, and the file content will be read according to the buffer defined in step (2) each time. In this way, we can avoid when reading large files, the computer's main storage area reads too much data, resulting in insufficient space and the system crashes.

2) Padding data block with zeros. The file is read multiple times according to the specified buffer. When the file is read for the last time, the size of the read file content is smaller than the buffer size. The read content needs to be filled with zeros to ensure that the read file data content can be equally divided into k blocks.

3) File data is divided into blocks. Each time the content of the read file needs to be divided into blocks to form a data block matrix D composed of multiple vectors to implement formula (4) and prepare for encoding to generate a check matrix.

4) Generate check matrix. Multiply the Vandermonde matrix in constructing the encoding matrix in Step (4) by D generated in step 3) to get the check matrix. Since the multiplication of the identity matrix by any matrix does not change the matrix multiplied by it, the check matrix can be obtained by directly multiplying the Vandermonde matrix by the equipartition data block matrix, avoiding unnecessary calculation. The equipartition data block matrix D combined with the generated check matrix is the encoding result of the data read this time [24].

5) Write the encoding result to the file. As shown in Algorithm 3, when writing the encoding result to a file, consider whether this is the first time to encode. When writing

the encoding result to a file for the first time, the file will be written in the 'wb' write mode. If the file does not exist, the file will be created, and the content of the encoding result will be written to the file; otherwise, the content of the encoding result will overwrite the original file's content. When writing the result of the non-first encoding to the file, the content of the encoding result can be directly added to the file. Write the encoding result into a file to convert each row vector of the encoding result matrix into a byte stream and write this byte stream into the corresponding file. The number of files after encoding is the same as the number of rows in the encoding result matrix(k+m) files. In addition, if the encoding process performed is the last time, that is, the encoding process performed after reading the file for the last time, after writing the encoding result into the file, the encoding ends. Otherwise, go to step 1) for the following encoding.

(6) Write a coding information file. After reading the file and encoding for the last time and writing the encoding result, write the encoded information to the encoded information file for decoding. The encoding information file stores the content, such as the file name, the number of equal-sized data blocks that the file is divided into, the number of redundancy check blocks, the buffer size, the times of reads, and the number of zero paddings.

## 3.4 Implementation Process of Data Decoding Based on RS Erasure Code

The pseudo-code of the file data decoding process is shown in Algorithm 4.

**Algorithm 4.** Pseudo-code of the decoding process

```
1.Enter a file name
2.Decoding preprocessing
3.Construct Galois field
4.Construct encoding matrix
5.Get the residual matrix
6.Find the inverse of the residual matrix
7.Get file read buffer
8.loop:
9.Get the remaining coding block matrix
10.Decode
11.if Decode for the non-last time then
12.  if Decode for the first time then
13.     Write file
14.  else
15.     Append to file
16.  end if
17.  goto loop.
18.else
19.  Remove zero padding part of decoding result
20.  if Decode for the first time then
21.     Write file
22.  else
23.     Append to file
24.  end if
25.end if
```

The data decoding process based on RS erasure code is shown in Figure 3.

The main steps of the data decoding implementation process based on the RS erasure code are as follows:

(1) Input file name. Input the file name to be decoded.

(2) Decoding preprocessing. Read the content of the corresponding encoded information file according to the input file name. Store the read content in the dictionary. Obtain any k remaining files numbers of non-failed data blocks including segmented data blocks and check redundant data blocks, and obtain the row numbers of the remaining vectors of the corresponding encoding matrix simultaneously.

(3) Construct Galois field.

(4) Construct encoding matrix.

(5) Obtain the residual matrix. According to the row number of the remaining vector acquired in step (2), the residual matrix $V'$ is obtained from the encoding matrix constructed in step (4).

(6) Invert the residual matrix. We invert the residual matrix obtained in step (5) to get $(V')^{-1}$.

(7) Get file read buffer. According to the buffer information in the encoded information obtained in step (2), the read file buffer is confirmed.

(8) Decoding. Decode according to the existing data and restore the source file. When encoding file data, the file is read and encoded multiple times, and the encoding results are combined and spliced into the encoding result file multiple times. Therefore, it is necessary to decode according to the logical structure of the file written when encoding. The main steps are as follows:

1) Obtain the remaining coding block matrix. Read the remaining files obtained in step (2). The file read buffer is obtained in step (7). The read buffer size corresponds to the buffer size when the file is encoded. The content of the read file is converted into a vector, and the remaining encoding block matrix $Y'$ is formed in the file reading order.

2) Decoding. On the basis of the Galois field and its multiplication, the inverse matrix $(V')^{-1}$ of the residual matrix of the encoding matrix obtained in step (6) is multiplied with the remaining coding block matrix $Y'$ obtained in step 1), that is, the realization of formula (9) can be obtained decoding result matrix.

3) Write decoding results to the file. As shown in Algorithm 4, it is necessary to consider whether this decoding is the last decoding when each decoding result is written to a file. If yes, it is necessary to remove the zero of the file data in the last encoding when encoding to ensure the accuracy of the decoding result. When the decoding result is written to the file, the writing mode 'wb' needs to be selected. The file is created if the first decoding result is written; otherwise, the writing content is overwritten. When the non-first decoding result is written to the file, it is directly added to the file; the file content is spliced. The times of decoding are the times of encodings during encoding, which can be obtained from the content of the encoding information file in step (2).

## 3.5 Numba and NumPy Accelerate PyRS

NumPy can provide vectors, matrices, and higher-dimensional data structures that perform better than Python. Its vectorization is implemented in C and Fortran code in intensive computing tasks, thus achieving high performance.

Numba is a speed optimization compiler for Python. It uses the industry-standard Low-Level Virtual Machine (LLVM) compiler library to convert Python functions into optimized machine code at runtime. Numba can instantly optimize Python code for arrays and math-heavy operations through simple comments, making its performance similar to

C, C++, and Fortran without switching languages or Python interpreters. In addition, Numba supports the NumPy library, so PyRS can use NumPy in the programs to speed up its overall calculations. Numba works by allowing Python functions to be specified with type signatures for "just-in-time" or JIT compilation at run time. The main steps of the Numba optimization are as follows:

Step 1: The Python function is passed into the Numba compile task, optimized, and converted to the Numba intermediate expressions.

Step 2: Type Interference is performed on the parameters of the Python function to determine the parameters' types.

Step 3: The Python function is converted to LLVM interpretable codes.

Step 4: The codes are provided to LLVM's just-in-time compiler to generate machine code.

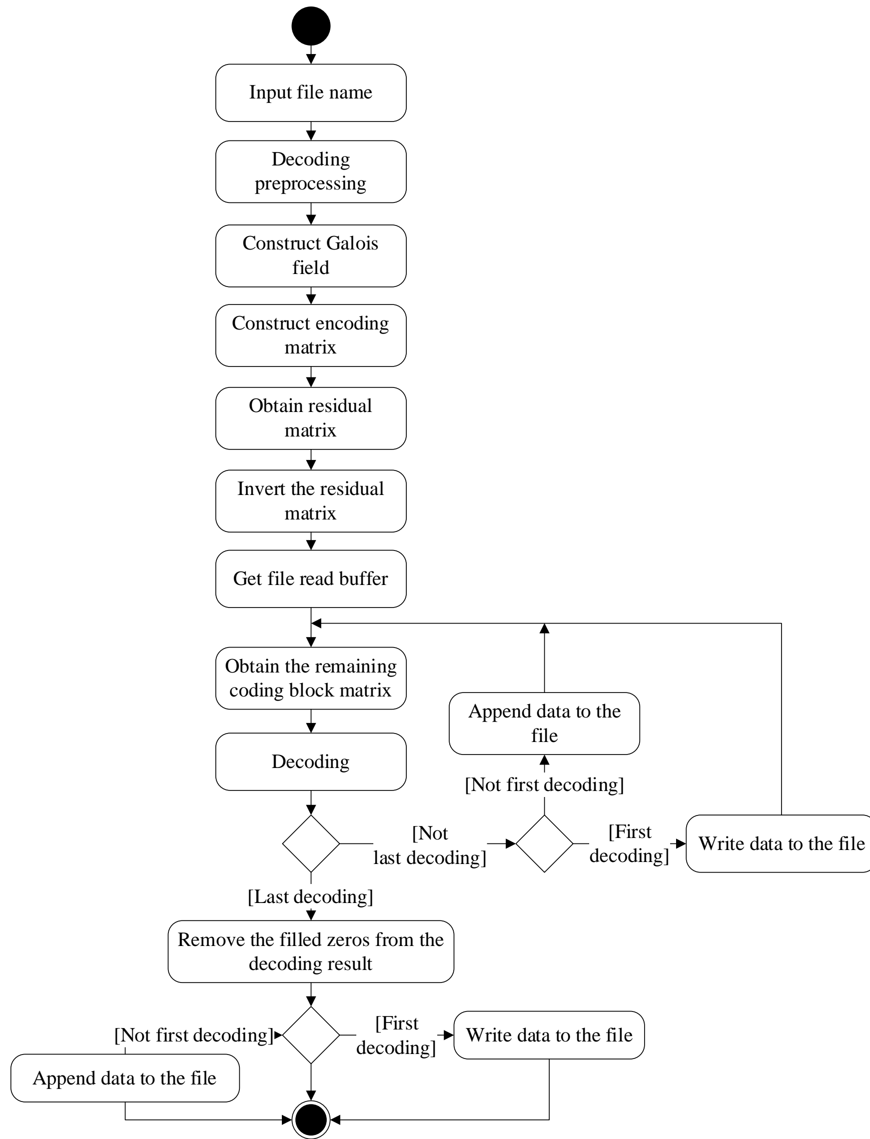



**Figure 3.** Activity diagram of data decoding process

# 4 Experiments

## 4.1 Machines for Experiments

We employed two machines for experimentation. Both machines are standard machines with 4 CPUs and 4GB of random memory. The first machine is a Dell workstation with an Intel(R) Core(TM) 4 CPU running at 2.30GHz with 4GB of RAM, a 32KB L1 cache, a 256KB L2 cache, and a 3MB L3 cache. The operating system is Linux version 3.10.0-957.el7.x86_64.

The second machine is a Lenovo workstation with Intel(R) Core(TM) 4 CPUs running at 2.20GHz with 4GB of RAM, an L1 cache of 32KB, an L2 cache of 256KB, and an L3 cache of 3MB. The operating system is Microsoft Windows 7 version 6.1.7601 Service Pack 1 Build 7601.

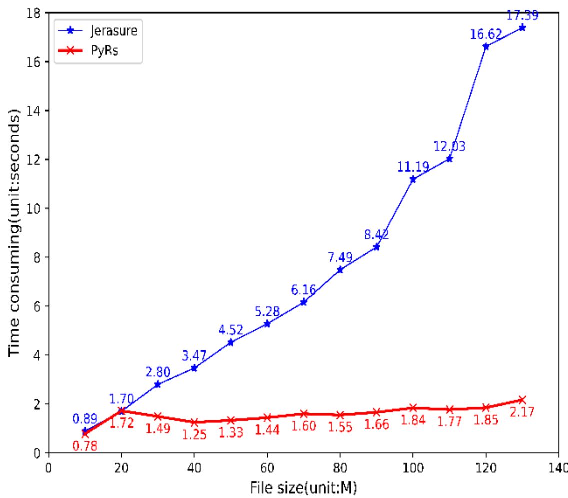
On either of the two machines, the encoder or decoder is executed while no other user programs are being executed.

## 4.2 Performance Comparison with Jerasure on Linux Platform

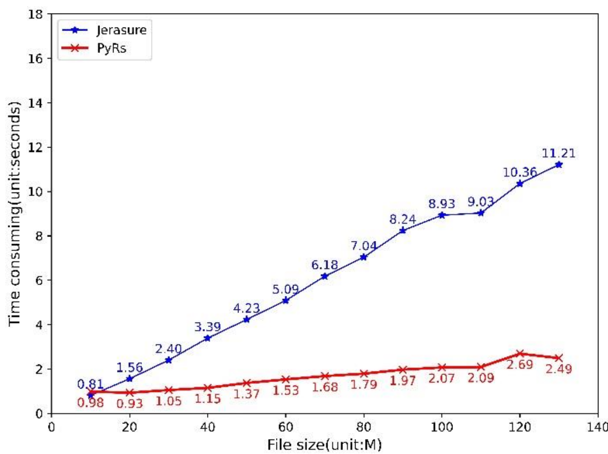
PyRS is compared with the Jerasure library whose execution environment is the Linux platform. To ensure the comparability and accuracy of the experimental data, both PyRS and Jerasure are tested on the Centos7 platform with 4G memory and a total number of processor cores of 4. The encoding parameters are buffer bufferSize=1024*1024*10 (10MB), number of file segmented blocks k=4, number of redundancy check blocks m=2.

### 4.2.1 Encoding/Decoding Speed Comparison

The encoding/decoding speed comparison experiment results of PyRS and Jerasure library are shown in Table 2. The line diagrams corresponding to Table 2 are shown in Figure 4.




(a) PyRS and Jerasure encoding time-consuming time




(b) PyRS and Jerasure decoding time-consuming time

**Figure 4.** PyRS and Jerasure time-consuming line diagrams on Linux platform

**Table 2.** Jerasure library and PyRS encoding/decoding time-consuming experiment results

| File size (MB) | Jerasure | | PyRS | |
|---|---|---|---|---|
| | Encoding (seconds) | Decoding (seconds) | Encoding (seconds) | Decoding (seconds) |
| 10 | 0.89 | 0.81 | 0.78 | 0.98 |
| 20 | 1.7 | 1.56 | 1.72 | 0.93 |
| 30 | 2.8 | 2.4 | 1.49 | 1.05 |
| 40 | 3.47 | 3.39 | 1.25 | 1.15 |
| 50 | 4.52 | 4.23 | 1.33 | 1.37 |
| 60 | 5.28 | 5.09 | 1.44 | 1.53 |
| 70 | 6.16 | 6.18 | 1.60 | 1.68 |
| 80 | 7.49 | 7.04 | 1.55 | 1.79 |
| 90 | 8.42 | 8.24 | 1.66 | 1.97 |
| 100 | 11.19 | 8.93 | 1.84 | 2.07 |
| 110 | 12.03 | 9.03 | 1.77 | 2.09 |
| 120 | 16.62 | 10.36 | 1.85 | 2.69 |
| 130 | 17.39 | 11.21 | 2.17 | 2.49 |

In Figure 4, the encoding/decoding time of Jerasure is significantly higher than that of PyRS, and with the increase of file size, the encoding/decoding time gap between Jerasure and PyRS keeps increasing.

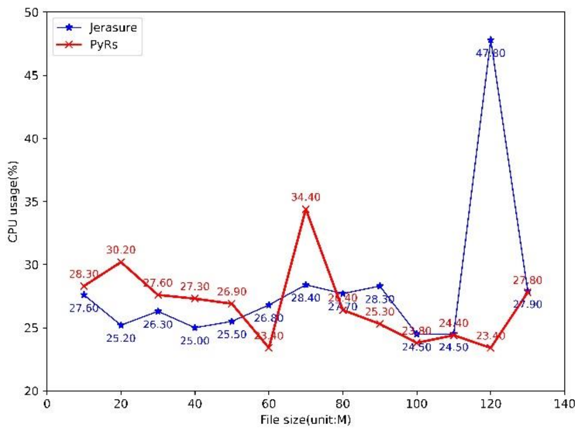
### 4.2.2 CPU Usage Comparison

PyRS compares CPU usage with Jerasure. In this paper, we design a program to collect the local CPU usage at an interval of 0.1 seconds when the file is being encoded or decoded. At the end of the file encoding or decoding, the collecting of CPU usage also follows end. The average value of CPU usage during encoding or decoding is taken as the CPU usage rate. The experimental results are shown in Table 3. The line diagrams corresponding to Table 3 are shown in Figure 5.

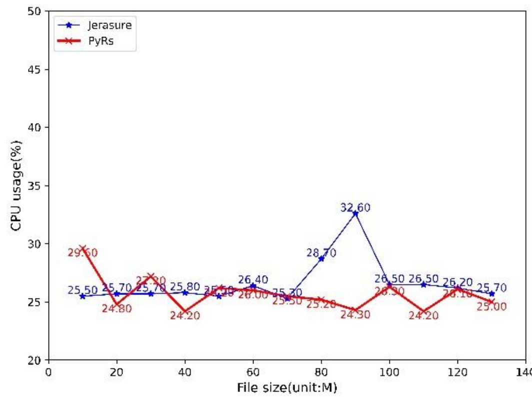
**Table 3.** CPU usage rate collection results of encoding/decoding of PyRS and Jerasure

| File size (MB) | Jerasure | | PyRS | |
|---|---|---|---|---|
| | Encoding | Decoding | Encoding | Decoding |
| 10 | 27.6% | 25.5% | 28.3% | 29.6% |
| 20 | 25.2% | 25.7% | 30.2% | 24.8% |
| 30 | 26.3% | 25.7% | 27.6% | 27.2% |
| 40 | 25.0% | 25.8% | 27.3% | 24.2% |
| 50 | 25.5% | 25.5% | 26.9% | 26.2% |
| 60 | 26.8% | 26.4% | 23.4% | 26.0% |
| 70 | 28.4% | 25.3% | 34.4% | 25.5% |
| 80 | 27.7% | 28.7% | 26.4% | 25.2% |
| 90 | 28.3% | 32.6% | 25.3% | 24.3% |
| 100 | 24.5% | 26.5% | 23.8% | 26.3% |
| 110 | 24.5% | 26.5% | 24.4% | 24.2% |
| 120 | 47.8% | 26.2% | 23.4% | 26.1% |
| 130 | 27.9% | 25.7% | 27.8% | 25.0% |

In Figure 5, bufferSize is the value set in PyRS and Jerasure library during encoding and decoding, so there is a certain amount of data in file reading and writing, encoding calculation, and decoding calculation of PyRS and Jerasure. Therefore, when file encoding and decoding, the CPU usages of PyRS and Jerasure are stable.




(a) PyRS and Jerasure encoding CPU usage line chart




(b) PyRS and Jerasure decoding CPU usage line chart

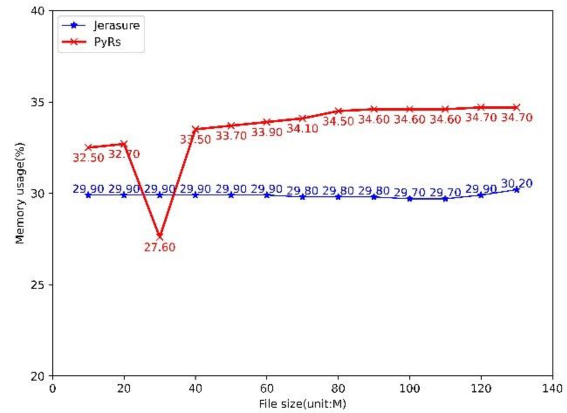
**Figure 5.** PyRS and Jerasure CPU usage line chart on Linux platform

### 4.2.3 Memory Usage Comparison

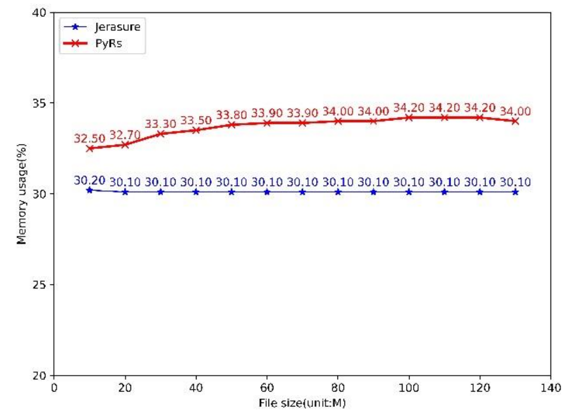
**Table 4.** Memory usage rate collection results of encoding/decoding of PyRS and Jerasure

| File | Jerasure | | PyRS | |
|---|---|---|---|---|
| size(MB) | Encoding | Decoding | Encoding | Decoding |
| 10 | 29.9% | 30.2% | 32.5% | 32.5% |
| 20 | 29.9% | 30.1% | 32.7% | 32.7% |
| 30 | 29.9% | 30.1% | 27.6% | 33.3% |
| 40 | 29.9% | 30.1% | 33.5% | 33.5% |
| 50 | 29.9% | 30.1% | 33.7% | 33.8% |
| 60 | 29.9% | 30.1% | 33.9% | 33.9% |
| 70 | 29.8% | 30.1% | 34.1% | 33.9% |
| 80 | 29.8% | 30.1% | 34.5% | 34.0% |
| 90 | 29.8% | 30.1% | 34.6% | 34.0% |
| 100 | 29.7% | 30.1% | 34.6% | 34.2% |
| 110 | 29.7% | 30.1% | 34.6% | 34.2% |
| 120 | 29.9% | 30.1% | 34.7% | 34.2% |
| 130 | 30.2% | 30.1% | 34.7% | 34.0% |

PyRS compares memory usage with the Jerasure library. In this paper, we design a program to collect the local memory usage at an interval of 0.1 seconds when the file is being encoded or decoded. At the end of the file encoding or decoding, the collecting of memory usage for the file encoding or decoding also follows the end. The average value of memory usage during encoding or decoding is taken as the memory usage rate. The experimental results are shown in Table 4. The line diagrams corresponding to Table 4 are shown in Figure 6.




(a) PyRS and Jerasure library encoding memory line chart




(b) PyRS and Jerasure library decoding memory line chart

**Figure 6.** PyRS and Jerasure memory usage line charts

In Figure 6, Jerasure and PyRS are limited by the parameter of bufferSize set before the encoding and decoding during the encoding and decoding process. The two kinds of memory usages are stable in the encoding and decoding process.

### 4.2.4 Experimental Analysis

In Table 2, compared with Jerasure library, the encoding/decoding time of PyRS is significantly less than that of Jerasure. When the file size is 120MB, the encoding/decoding time of Jerasure library is 8 times as much as that of PyRS. Therefore, PyRS performs better than Jerasure in terms of speed. For Jerasure, the encoding time is always greater than the decoding time, and when the file size is less than 90MB, the encoding and decoding time are very close. As the file size increases, the ratio of encoding and decoding time is always larger. When the file size is 120MB, the ratio is 1.6. But for PyRS, the encoding time is less than the decoding time in most cases, and the two values are very close, and the difference between them is very small. And with the increase of file size, the encoding time is less than the decoding time. The ratio of encoding and decoding time does not increase with the increase of file size, and the value of the ratio changes little.

In Table 3, Jerasure has a significant change in CPU usage during encoding, but the CPU usage is equivalent to PyRS generally. In decoding, the CPU usage of PyRS is almost equal to Jerasure. The CPU usage remains around 25% in most cases. PyRS and Jerasure library have almost the same CPU resource usage. For Jerasure and PyRS, the difference between the encoding and decoding CPU usage rate is small.

In Table 4, the encoding and decoding memory usage rates of PyRS are almost equivalent, as is the Jerasure library. Compared with Jerasure, the memory usage rate of PyRS is slightly larger, but for 4G memory, this gap can be ignored. For Jerasure, the system memory usage rate is maintained at about 30%. For PyRS, the system memory usage rate is maintained at about 35%. Therefore, the memory load of PyRS is not high. Compared with Jerasure, PyRS has a little higher memory usage rate in the encoding and decoding process, with a difference of about 5%. For Jerasure and PyRS, the difference between the encoding and decoding memory usage rate is small.

## 4.3 Test PyRS on the Windows Platform

To compare the performance of PyRS on Linux and Windows platforms, PyRS is tested on the Windows7 platform with the same memory of 4G and a total number of processor cores of 4. The experimental results of PyRS on the Centos7 platform in Section 5.2.1 are selected as the experimental results of PyRS on the Linux platform. To ensure the comparability of the experimental results, PyRS is tested on the Windows7 platform by choosing the same coding parameters as the Centos7 platform test in Section 5.2.1.

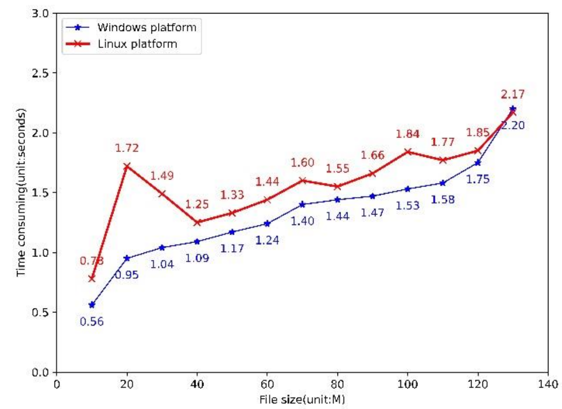
### 4.3.1 Encoding/Decoding Speed Comparison

Table 5 compares the encoding and decoding time-consuming of PyRS on the Linux and Windows platforms. The corresponding line chart in Table 5 is shown in Figure 7.

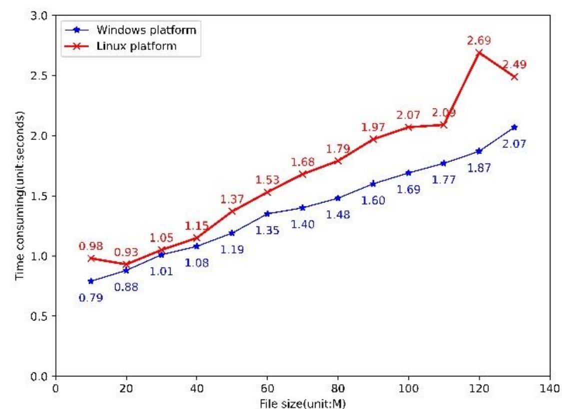
**Table 5.** Linux platform and Windows platform, PyRS encoding/decoding Time-Consuming

| File size (MB) | Windows | | Linux | |
|---|---|---|---|---|
| | Encoding (seconds) | Decoding (seconds) | Encoding (seconds) | Decoding (seconds) |
| 10 | 0.56 | 0.79 | 0.78 | 0.98 |
| 20 | 0.95 | 0.88 | 1.72 | 0.93 |
| 30 | 1.04 | 1.01 | 1.49 | 1.05 |
| 40 | 1.09 | 1.08 | 1.25 | 1.15 |
| 50 | 1.17 | 1.19 | 1.33 | 1.37 |
| 60 | 1.24 | 1.35 | 1.44 | 1.53 |
| 70 | 1.40 | 1.40 | 1.60 | 1.68 |
| 80 | 1.44 | 1.48 | 1.55 | 1.79 |
| 90 | 1.47 | 1.60 | 1.66 | 1.97 |
| 100 | 1.53 | 1.69 | 1.84 | 2.07 |
| 110 | 1.58 | 1.77 | 1.77 | 2.09 |
| 120 | 1.75 | 1.87 | 1.85 | 2.69 |
| 130 | 2.20 | 2.07 | 2.17 | 2.49 |

In Figure 7, the encoding and decoding time-consuming of PyRS on the Windows and Linux platforms have a small gap of milliseconds, which can be ignored.




(a)PyRS encoding time-consuming line chart




(b)PyRS decoding time-consuming line chart

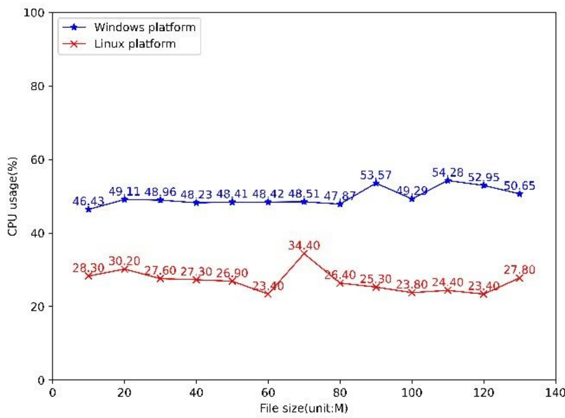
**Figure 7.** PyRS encoding and decoding time-consuming line charts on the Windows and Linux platforms

### 4.3.2 CPU Usage Comparison

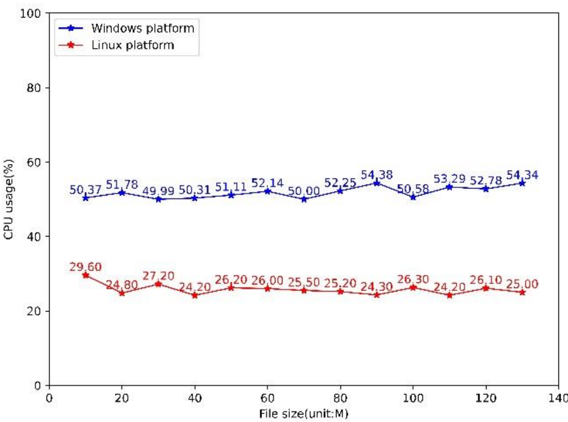
The same method used in Sections 5.2.2 is adopted to collect the CPU usage rate during PyRS encoding and decoding, and the collection results are shown in Table 6. Figure 8 is the corresponding line diagram.

**Table 6.** CPU usage rate of PyRS encoding/decoding on Windows and Linux platforms

| File size (MB) | Windows | | Linux | |
|---|---|---|---|---|
| | Encoding | Decoding | Encoding | Decoding |
| 10 | 46.43% | 50.37% | 28.3% | 29.6% |
| 20 | 49.11% | 51.78% | 30.2% | 24.8% |
| 30 | 48.96% | 49.99% | 27.6% | 27.2% |
| 40 | 48.23% | 50.31% | 27.3% | 24.2% |
| 50 | 48.41% | 51.11% | 26.9% | 26.2% |
| 60 | 48.42% | 52.14% | 23.4% | 26.0% |
| 70 | 48.51% | 50.00% | 34.4% | 25.5% |
| 80 | 47.87% | 52.25% | 26.4% | 25.2% |
| 90 | 53.57% | 54.38% | 25.3% | 24.3% |
| 100 | 49.29% | 50.58% | 23.8% | 26.3% |
| 110 | 54.28% | 53.29% | 24.4% | 24.2% |
| 120 | 52.95% | 52.78% | 23.4% | 26.1% |
| 130 | 50.65% | 54.34% | 27.8% | 25.0% |

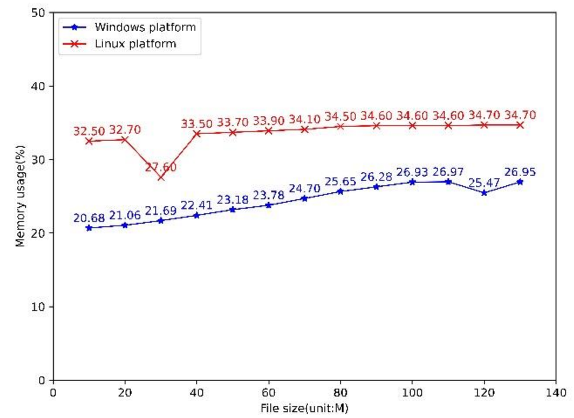


(a) Line chart of CPU usage of PyRS encoding

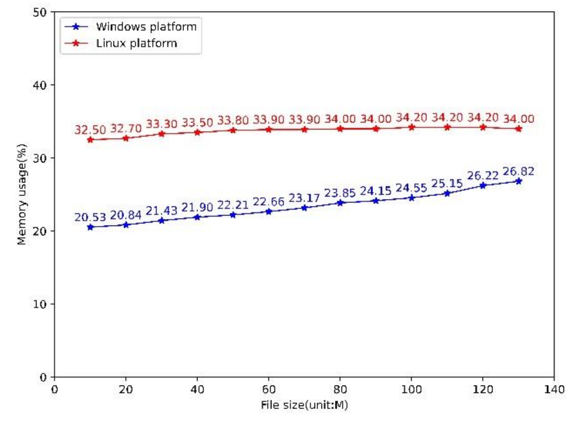


(b) Line chart of CPU usage of PyRS decoding

**Figure 8.** Line diagram of PyRS CPU usage on Windows and Linux platforms



(a) Line chart of Memory usage of PyRS encoding



(b) Line chart of Memory usage of PyRS decoding

**Figure 9.** Line diagram of PyRS Memory usage on Windows and Linux platforms

### 4.3.3 Memory Usage Comparison

The same method used in Sections 4.2.3 is adopted to collect the system memory usage rate during PyRS encoding and decoding, and the collection results are shown in Table 7. Figure 9 is the corresponding line diagram.

**Table 7.** Memory usage rate of PyRS encoding and decoding on Windows and Linux platforms

| File size | Windows | | Linux | |
|---|---|---|---|---|
| (MB) | Encoding | Decoding | Encoding | Decoding |
| 10 | 20.68% | 20.53% | 32.5% | 32.5% |
| 20 | 21.06% | 20.84% | 32.7% | 32.7% |
| 30 | 21.69% | 21.43% | 27.6% | 33.3% |
| 40 | 22.41% | 21.90% | 33.5% | 33.5% |
| 50 | 23.18% | 22.21% | 33.7% | 33.8% |
| 60 | 23.78% | 22.66% | 33.9% | 33.9% |
| 70 | 24.70% | 23.17% | 34.1% | 33.9% |
| 80 | 25.65% | 23.85% | 34.5% | 34.0% |
| 90 | 26.28% | 24.15% | 34.6% | 34.0% |
| 100 | 26.93% | 24.55% | 34.6% | 34.2% |
| 110 | 26.97% | 25.15% | 34.6% | 34.2% |
| 120 | 25.47% | 26.22% | 34.7% | 34.2% |
| 130 | 26.95% | 26.82% | 34.7% | 34.0% |

### 4.3.4 Experimental Analysis

In Table 5, for PyRS on Windows, the encoding and the decoding time are very close, and the difference between them is very small. And the ratio of encoding and decoding time does not increase with the increase of file size, and the value of the ratio changes very little. The speed of PyRS on the Windows platform and the Linux platform is almost equivalent.

Table 6 shows that the CPU usage rate of PyRS encoding and decoding on the Windows platform is higher than it on Linux. The system CPU usage rate of PyRS on Windows is maintained at about 50%, but it is about 25% on Linux. Table 7 shows that the memory usage rate of PyRS encoding and decoding on the Windows platform is not high. The system memory usage rate is maintained at about 25% on Windows, but it is about 35% on Linux. Both on Windows and Linux, the system CPU and memory usage is stable and is not affected by the file size.

## 5 Conclusion

This paper designs and implements an Erasure Code library PyRS based on RS Erasure Code for Windows and Linux platforms. PyRS, developed in Python, has the

advantages of cross-platform, simple operation, and easy environment configuration. In addition, PyRS ensures the accuracy of the data through the Galois field and speeds up encoding and decoding using Numba and NumPy libraries. After the test, PyRS is up to 8 times faster than Jerasure in the same Linux environment. PyRS has a CPU usage comparable to Jerasure. Compared to Jerasure, PyRS has a higher memory usage with a memory footprint difference of about 5%. And it can be seen from the experimental results that PyRS has the same speed on the Windows platform as on the Linux platform, and the CPU and memory load of PyRS is not high.

Future research will focus on supporting 16-bit or 32-bit words in the Galois field, using the Cauchy matrix to construct a generation matrix for encoding and decoding, and storing encrypted k+m data blocks in a distributed system or multi-cloud storage system.

## Acknowledgment

This work was supported by Foundation of Hebei University of Science and Technology under Grant 2019-ZDB02.

## References

[1] J. S. Plank, Erasure codes for storage systems: A brief primer, *login: the magazine of USENIX & SAGE*, Vol. 38, No. 6, pp. 44-50, December, 2013.

[2] L. Yang, L. Ma, C. A. Lv, Y. Li, M. Shang, F. Tong, On Dynamic Replication Strategies of HDFS Cloud Storage Based on RS Erasure Codes, *Technology Innovation and Application*, Vol. 8, No. 24, pp. 38-39, August, 2018.

[3] H. Weatherspoon, J. D. Kubiatowicz, Erasure Coding Vs. Replication: A Quantitative Comparison, *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, USA, 2002, pp. 328-337.

[4] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, Erasure Coding in Windows Azure Storage, *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, Boston, MA, USA, 2012, pp. 1-12.

[5] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, S. Kumar, f4: Facebook's Warm BLOB Storage System, *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, USA, 2014, pp. 383-398.

[6] J. S. Plank, S. Simmerman, C. D. Schuman, *Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications Version 1.2*, Technical Report CS-08-627, August, 2008.

[7] X. Chu, C. Liu, K. Ouyang, L. S. Yung, H. Liu, Y.-W. Leung, PErasure: a Parallel Cauchy Reed-Solomon Coding Library for GPUs, *IEEE International Conference on Communications (ICC)*, London, UK, 2015, pp. 436-441.

[8] S. Dai, X. Li, CUDA-based Performance Optimization of RS Erasure Coding, *Microcomputer Applications*, Vol. 32, No. 1, pp. 70-72, January, 2016.

[9] L. Zhou, *The Research and Application on Reed-Solomon Codes Based on Distributed Storage System*, Master's Thesis, Chengdu University of Technology, Chengdu, China, 2017.

[10] X. Gong, C. W. Sung, Zigzag Decodable codes: Linear-time erasure codes with applications to data storage, *Journal of Computer and System Sciences*, Vol. 89, pp. 190-208, November, 2017.

[11] C. Liu, Q. Wang, X. Chu, Y.-W. Leung, G-CRS: GPU Accelerated Cauchy Reed-Solomon Coding, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 29, No. 7, pp. 1484-1498, July, 2018.

[12] W. Liu, N. Zhou, X. Gao, L-Code: An Efficient Coding Scheme for Recovering Single Disk Failure, *2018 8th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Beijing, China, 2018, pp. 108-111

[13] X. Xie, C. Wu, J. Gu, H. Qiu, J. Li, M. Guo, X. He, Y. Dong, Y. Zhao, AZ-Code: An Efficient Availability Zone Level Erasure Code to Provide High Fault Tolerance in Cloud Storage Systems, *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, Santa Clara, CA, USA, 2019, pp. 230-243.

[14] H. Qiu, C. Wu, J. Li, M. Guo, T. Liu, X. He, Y. Dong, Y. Zhao, EC-Fusion: An Efficient Hybrid Erasure Coding Framework to Improve Both Application and Recovery Performance in Cloud Storage Systems, *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, New Orleans, LA, USA, 2020, pp. 191-201.

[15] Y. Fang, S. Wang, H. Tan, X. Zhang, J. Zhang, CLRC: a New Erasure Code Localization Algorithm for HDFS, *2021 International Conference on Computer Engineering and Artificial Intelligence (ICCEAI)*, Shanghai, China, 2021, pp. 62-65.

[16] S. S. Arslan, Founsure 1.0: An erasure code library with efficient repair and update features, *SoftwareX*, Vol. 13, pp. 1-12, January, 2021.

[17] X. Li, R. Sun, J. Liu, Overview of Fault-tolerant Techniques in Distributed Storage Systems, *Radio Communications Technology*, Vol. 45, No. 5, pp. 463-475, July, 2019.

[18] P. Li, J. Li, R. J. Stones, G. Wang, Z. Li, X. Liu, ProCode: A Proactive Erasure Coding Scheme for Cloud Storage Systems, *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, Budapest, Hungary, 2016, pp. 219-228.

[19] R. Li, X. Li, P. P. C. Lee, Q. Huang, Repair Pipelining for Erasure-Coded Storage, *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, Santa Clara, CA, USA, 2017, pp. 567-579.

[20] L. Zheng, X. Li, Low-cost Multi-node Failure Repair Method for Erasure Codes, *Computer Engineering*, Vol. 43, No. 7, pp. 110-118+123, July, 2017.

[21] L. Sun, Y. Su, C. Zhang, T. Zhang, Research on Fault-Tolerant Method of Erasure Code for Distributed Storage System, *Computer Engineering*, Vol. 45, No. 11, pp. 74-80, November, 2019.

[22] Y. Tang, F. Wang, Y. Xie, An Efficient Failure Reconstruction Based on In-Network Computing for Erasure-Coded Storage Systems, *Journal of Computer Research and Development*, Vol. 56, No. 4, pp. 767-778, April, 2019.

[23] I. S. Reed, G. Solomon, Polynomial Codes over Certain Finite Fields, *Journal of the Society for Industrial and*

*Applied Mathematics*, Vol. 8, No. 2, pp. 300-304, June, 1960.

[24] Y. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, D. Phillips, Giza: Erasure Coding Objects across Global Data Centers, *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, Santa Clara, USA, 2017, pp. 539-551.

## Biographies

**Junqiang Ma** is currently a lecturer with Department of Electrical Engineering, Hebei Vocational University of Technology and Engineering, Xingtai Hebei, China. His research interest is computer control technology.

**Weihao Yan** is currently pursuing the master's degree with the Hebei University of Science and Technology. His research interests include data processing and security, image detection.

**Xiaotian Zhang** is currently pursuing the master's degree with the Hebei University of Science and Technology. His research interests include machine learning and deep learning.

**Min Huang** is currently an associate professor with School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include machine learning, data processing and security, artificial intelligence.

**Jingyang Wang** is currently a Professor with School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include machine learning, big data processing and distributed system.