Exact Regenerating Codes for Byzantine Fault Tolerance in Distributed Storage

Yunghsiang S. Han
Dept. of Electrical Engineering
National Taiwan Univ. of Sci. and Tech.
Taiwan, R.O.C.
E-mail: *yshan@mail.ntust.edu.tw*

Rong Zheng
Dept. of Computer Science
University of Houston
Houston, TX
E-mail: *rzheng@uh.edu*

Wai Ho Mow
Dept. of Electronic & Computer Eng.
Hong Kong Univ. of Sci. and Tech.
Hong Kong
E-mail: *w.mow@ieee.org*

***Abstract*—Due to the use of commodity software and hardware, crash-stop and Byzantine failures are likely to be more prevalent in today's large-scale distributed storage systems. Regenerating codes have been shown to be a more efficient way to disperse information across multiple nodes and recover crash-stop failures in the literature. In this paper, we present the design of regeneration codes in conjunction with integrity check that allows exact regeneration of failed nodes and data reconstruction in the presence of Byzantine failures. A progressive decoding mechanism is incorporated in both procedures to leverage computation performed thus far. The fault tolerance and security properties of the schemes are also analyzed.**

***Index Terms*—Network storage, Regenerating code, Byzantine failures, Reed-Solomon code, Error-detection code**

I. Introduction

Storage is becoming a commodity due to the emergence of new storage media and the ever decreasing cost of conventional storage devices. Reliability, on the other hand, continues to pose challenges in the design of large-scale distributed systems such as data centers. Today's data centers operate on commodity hardware and software, where both crash-stop and Byzantine failures (as a result of software bugs, attacks) are likely to be the norm. To achieve persistent storage, one common approach is to disperse information pertaining to a data file (the message) across nodes in a network. For instance, with (n, k) maximum-distance-separable (MDS) codes such as Reed-Solomon (RS) codes, data are encoded and stored across n nodes and, an end user or a data collector can retrieve the original data file by accessing *any k* of the storage nodes, a process referred to as *data reconstruction*.

Upon failure of any storage node, data stored in the failed node need to be regenerated (recovered) to maintain the functionality of the system. A straightforward way for data recovery is to first reconstruct the original data and then regenerate the data stored in the failed node. However, it is wasteful to retrieve the entire B symbols of the original file, just to recover a small fraction of that stored in the failed node. A more efficient way is to use the *regenerating codes* which was introduced in the pioneer works by Dimakis *et al.* in [1], [2]. A trade-off can be made between the storage overhead and the repair bandwidth needed for regeneration. Minimum Storage Regenerating (MSR) codes minimize first, the amount of data stored per node, and then the repair bandwidth, while Minimum Bandwidth Regenerating (MBR) codes carry out the minimization in the reverse order. The design of regenerating codes have received much attention in recent years [3]–[10]. Most notably, Rashmi *et al.* proposed optimal exact-regenerating codes using a product-matrix reconstruction that recovers exactly the same stored data of the failed node (and thus the name exact-regenerating) [10]. Existing work assumes crash-stop behaviors of storage nodes. However, with Byzantine failures, the stored data may be tampered resulting in erroneous data reconstruction and regeneration.

In this paper, we consider the problem of exact regeneration for Byzantine fault tolerance in distributed storage networks. Two challenging issues arise when nodes may fail arbitrarily. First, we need to verify whether the regenerated or reconstructed data are correct. Second, efficient algorithms are needed that *incrementally* retrieve additional stored data and perform data reconstruction and regeneration when errors have been detected. We adopt cyclic redundancy codes (CRC) to verify the integrity of stored data. In particular, for data reconstruction, CRC is coded along with the original data and distributed among storage nodes. For data regeneration, checksums for each storage node are stored distributively to ensure the correctness of verification in face of node failures. Incremental retrieval, reconstruction and regeneration are made possible by the use of a progressive decoding procedure.

Our work is inspired by [10] and makes the following new contributions:

- We present the detailed design of an exact-regenerating code with error correction capability.[1]
- We devise a procedure that verifies the correctness of regenerated/reconstructed data.
- We propose progressive decoding algorithms for data reconstruction and regeneration that leverage computation performed thus far.

The rest of the paper is organized as follows. We give an overview of regenerating codes and RS codes in Section II to

This work was supported in part by National Science Council, R.O.C., under grant NSC-99-2221-E-011-157, the AoE Grant E-02/08 from the University Grants Committee of the Hong Kong Special Administration Region, China, and US National Science Foundation under contract #0546391.

[1]The encoding process is the same as that given in [10] except that an explicit encoding matrix is given in this work.

prepare the readers with necessary background. The design of error-correcting exact regenerating code for the MSR points and MBR points are presented in Section III and Section IV, respectively. Analytical results on the fault tolerance and security properties of the proposed schemes are given in Section V. Related work is briefly surveyed in Section VI. Finally, we conclude the paper in Section VII.

II. Preliminaries

A. Regenerating Codes

Regenerating codes achieve bandwidth efficiency in the regeneration process by storing additional symbols in each storage node or accessing more storage nodes. Let α be the number of symbols over finite field $GF(q)$ stored in each storage node and $\beta \leq \alpha$ the number of symbols downloaded from each storage during regeneration. To repair the stored data in the failed node, a helper node accesses d surviving nodes with the total repair bandwidth $d\beta$. In general, the total repair bandwidth is much less than B. A regenerating code can be used not only to regenerate coded data but also to reconstruct the original data symbols. Let the number of storage nodes be n. An $[n,k,d]$ regenerating code requires at least k and d surviving nodes to ensure successful data reconstruction and regeneration [10], respectively. Clearly, $k \leq d \leq n-1$.

The main results given in [2], [3] are the so-called cut-set bound on the repair bandwidth. It states that any regenerating code must satisfy the following inequality:

$$B \leq \sum_{i=0}^{k-1} \min\{\alpha, (d-i)\beta\} \ . \quad (1)$$

Minimizing α in (1) results in a regenerating code with minimum storage requirement; and minimizing β results in that with minimum repair bandwidth. It is impossible to have minimum values both on α and β concurrently, and thus there exists a trade-off between storage and repair bandwidth. The two extreme points in (1) are referred to as the minimum storage regeneration (MSR) and minimum bandwidth regeneration (MBR) points, respectively. The values of α and β for MSR point can be obtained by first minimizing α and then minimizing β:

$$\begin{aligned} \alpha &= \frac{B}{k} \\ \beta &= \frac{B}{k(d-k+1)} \ . \end{aligned} \quad (2)$$

Reversing the order of minimization we have β and α for MBR as

$$\begin{aligned} \beta &= \frac{2B}{k(2d-k+1)} \\ \alpha &= \frac{2dB}{k(2d-k+1)} \ . \end{aligned} \quad (3)$$

As defined in [10], an $[n,k,d]$ regenerating code with parameters (α,β,B) is optimal if i) it satisfies the cut-set bound with equality, and ii) neither α and β can be reduced unilaterally without violating the cut-set bound. Clearly, both MSR and MBR codes are optimal regenerating codes.

It has been proved that when designing $[n,k,d]$ MSR for $k/(n+1) \leq 1/2$ or MBR codes, it suffices to consider those with $\beta = 1$ [10]. Throughout this paper, we assume that $\beta = 1$ for code design. Hence (2) and (3) become

$$\begin{aligned} \alpha &= d-k+1 \\ B &= k(d-k+1) = k\alpha \end{aligned} \quad (4)$$

and

$$\begin{aligned} \alpha &= d \\ B &= kd - k(k-1)/2 \ , \end{aligned} \quad (5)$$

respectively, when $\beta = 1$.

There are two ways to regenerate data for a failed node. If the replacement data generated is exactly the same as those stored in the failed node, we call it the *exact regeneration*. If the replacement data generated is only to guarantee the data reconstruction and regeneration properties, it is called *functional regeneration*. In practice, exact regeneration is more desired since there is no need to inform each node in the network regarding the replacement. In addition, it is easy to keep the code systematic via exact regeneration, where partial data can be retrieved without accessing k nodes. Throughout this paper, we only consider exact regeneration and design exact-regenerating codes with error-correction capabilities. We target for all possible n,k for the MBR points and $k/(n+1) \leq 1/2$ for the MSR points.

B. Reed-Solomon codes

Since Reed-Solomon (RS) codes will be used in the design of regenerating codes, we briefly describe the encoding and decoding mechanisms of RS codes next.

RS codes are the most well-known error-correction codes. They not only can recover data when nodes fail, but also can guarantee recovery when a subset of nodes are Byzantine. RS codes operate on symbols of m bits, where all symbols are from finite field $GF(2^m)$. An $[n,d]$ RS code is a linear code, with parameters $n = 2^m - 1$ and $n - d = 2t$, where n is the total number of symbols in a codeword, d is the total number of information symbols, and t is the symbol-error-correction capability of the code.

Encoding: Let the sequence of d information symbols in $GF(2^m)$ be $\boldsymbol{u} = [u_0, u_1, \ldots, u_{d-1}]$ and $u(x)$ be the information polynomial of $\boldsymbol{u}$ represented as[2]

$$u(x) = u_0 + u_1 x + \cdots + u_{d-1}x^{d-1} \ .$$

The codeword polynomial, $c(x)$, corresponding to $u(x)$ can be encoded as [11]

$$c(x) = u(a^0) + u(a^1)x + u(a^2)x^2 + \cdots + u(a^{n-1})x^{n-1} \ . \quad (6)$$

[2]We use polynomial and vectorized representations of information symbols, codewords, received symbols and errors interchangeably in this work.

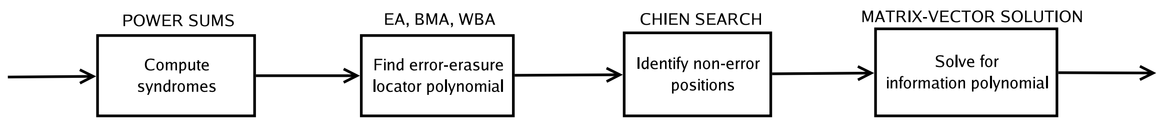
POWER SUMS
Compute syndromes
EA, BMA, WBA
Find error-erasure locator polynomial
CHIEN SEARCH
Identify non-error positions
MATRIX-VECTOR SOLUTION
Solve for information polynomial

Fig. 1. Block diagram of RS decoding. Above each block, the corresponding existing algorithms are indicated.

Let

$$\begin{aligned} g(x) &= (x-a)(x-a^2)\cdots(x-a^{2t}) \\ &= g_0 + g_1 x + g_2 x^2 + \cdots + g_{2t}x^{2t} \ , \end{aligned} \quad (7)$$

where a is a generator (or a primitive element) in $GF(2^m)$ and $g_i \in GF(2^m)$. It can be proved that $c(x)$ is divisible by $g(x)$, i.e., $a, a^2, \ldots, a^{2t}$ are roots of $c(x)$.

Decoding: The decoding process of RS codes is more complex. A complete description can be found in [12].

Let $r(x)$ be the received polynomial and $r(x) = c(x) + e(x) + \gamma(x) = c(x) + \lambda(x)$, where $e(x) = \sum_{j=0}^{n-1} e_j x^j$ is the error polynomial, $\gamma(x) = \sum_{j=0}^{n-1} \gamma_j x^j$ the erasure polynomial, and $\lambda(x) = \sum_{j=0}^{n-1} \lambda_j x^j = e(x) + \gamma(x)$ the errata polynomial. Note that $g(x)$ and (hence) $c(x)$ have $a, a^2, \ldots, a^{2t}$ as roots. This property is used to determine the error locations and recover the information symbols.

The RS codes are optimal in terms of minimum Hamming distance as it meets the Singleton bound [12]. An $[n, d]$ RS code can recover from any v errors as long as $v \le \lfloor \frac{n-d-s}{2} \rfloor$, where s is the number of erasure (or irretrievable symbols). The basic procedure of RS decoding is shown in Figure 1. The last step in this figure is not necessary if a systematic RS code is applied; otherwise, the last step of the decoding procedure involves solving a set of linear equations, and can be made efficient by the use of Vandermonde generator matrices [13]. The decoding that handles both error and erasure is called the error-erasure decoding.

In $GF(2^m)$, addition is equivalent to bit-wise exclusive-or (XOR), and multiplication is typically implemented with multiplication tables or discrete logarithm tables. To reduce the complexity of multiplication, Cauchy Reed-Solomon (CRS) codes [14] have been proposed to use a different construction of the generator matrix, and convert multiplications to XOR operations for erasure. However, CRS codes incur the same complexity as RS codes for error correction.

III. Encoding and Decoding of Error-Correcting Exact-Regenerating Codes for the MSR Points

In this section, we demonstrate how to perform error correction on MSR codes designed to handle Byzantine failures by extending the code construction in [10]. It has been proved in [10] that an MSR code $\boldsymbol{C}'$ with parameters $[n', k', d']$ for any $2k'-2 \le d' \le n'-1$ can be constructed from an MSR code $\boldsymbol{C}$ with parameters $[n = n'+i, k = k'+i, d = d'+i]$, where $d = 2k-2$ and $i = d' - 2k' + 2$. Furthermore, if $\boldsymbol{C}$ is linear, so is $\boldsymbol{C}'$. Hence, it is sufficient to design an MSR code for $d = 2k-2$. Since $d \le n-1$ it is clear that $2k-2 \le n-1$ and then $k/(n+1) \le 1/2$. When $d = 2k-2$ we have

$$\alpha = d - k + 1 = k - 1 = d/2$$

and

$$B = k\alpha = \alpha(\alpha+1) \ .$$

We assume that the symbols in data are elements from $GF(2^m)$. Hence, the size of the data is mB bits for $\beta = 1$.

A. Verification for Data Reconstruction

Since we need to design codes with Byzantine fault tolerance it is necessary to perform integrity check after the original data are reconstructed. Two common verification mechanisms can be used: CRC and hash function. Both methods add redundancy to the original data before they are encoded. Here we adopt CRC since it is simple to implement and requires less redundancy.

CRC uses a cyclic code (CRC code) such that each information sequence can be verified using its generator polynomial with degree r, where r is the number of redundant bits added to the information sequence [12], [15]. The amount of errors that can be detected by a CRC code is related to the number of redundant bits. A CRC code with r redundant bits *cannot* detect $(\frac{1}{2^r}) \times 100\%$ portion of errors or more. For example, when $r = 32$, the mis-detection error probability is on the order of 10^{-10}. Since the size of the original data is usually large, the redundancy added by imposing a CRC code is relatively small. For example, for a $[100, 20, 38]$ MSR code with $\alpha = 19$, $B = 19 \times 20 = 380$, we need to operate on $GF(2^{11})$ such that the size of the original data is 4180 bits. If $r = 32$, then only 0.77% redundancy is added. Hence, in the following, we assume that the CRC checksum has been added to the original data and the resultant size is B symbols.

In order to enhance the strength of verification in some applications, cryptographic hash function [16] can be considered. A cryptographic hash function is a one-way function that takes an arbitrary block of data and returns a fixed-size binary string, namely, the hash value. Modifying the input data without changing the hash value is computationally infeasible. Furthermore, it is also infeasible to generate the input data given the hash value alone. The design of cryptographic hash functions is usually based on block ciphers such as AES. Since the length of input data of a block cipher is fixed to some numbers, the Merkle-Damgård construction is adopted [17], [18]. A common length of the hash value is 128 bits, which is much longer than a CRC checksum. However, the operation of a cryptographic hash function is more complicated than that of CRC.

B. Encoding

We arrange the information sequence $\boldsymbol{m} = [m_0, m_1, \ldots, m_{B-1}]$ into an information vector U with size $\alpha \times d$ such that

$$U = [A_1 A_2]$$

where

$$A_1 = \begin{bmatrix} m_0 & m_1 & m_2 & \cdots & m_{\alpha-1} \\ m_1 & m_\alpha & m_{\alpha+1} & \cdots & m_{2\alpha-2} \\ & & \vdots & & \\ m_{\alpha-1} & m_{2\alpha-2} & m_{3\alpha-4} & \cdots & m_{\alpha(\alpha+1)/2-1} \end{bmatrix}$$

and

$$A_2 = \begin{bmatrix} m_{\alpha(\alpha+1)/2} & m_{\alpha(\alpha+1)/2+1} & \cdots & m_{\alpha(\alpha+1)/2+\alpha-1} \\ m_{\alpha(\alpha+1)/2+1} & m_{\alpha(\alpha+1)/2+\alpha} & \cdots & m_{\alpha(\alpha+1)/2+2\alpha-2} \\ & & \vdots & \\ m_{\alpha(\alpha+1)/2+\alpha-1} & m_{\alpha(\alpha+1)/2+2\alpha-2} & \cdots & m_{B-1} \end{bmatrix}$$

Precisely, we have

$$u_{ij} = \begin{cases} u_{ji} = m_{k_1} & \text{for } i \leq j \leq \alpha \\ u_{(j-\alpha)i} = m_{k_2} & \text{for } i+\alpha \leq j \leq 2\alpha \end{cases},$$

where $k_1 = (i-1)(\alpha+1) - i(i+1)/2 + j$ and $k_2 = (\alpha+1)(i-1+\alpha/2) - i(i+1)/2 + (j-\alpha)$. From the above construction, A_j's are symmetric matrices with dimension $\alpha \times \alpha$ for $j = 1, 2$.

In this encoding, each row of the information vector U produces a codeword of length n. An $[n, d = 2\alpha]$ RS code is adopted to construct the MSR code. In particular, for the ith row of U, the corresponding codeword is

$$[p_i(a^0 = 1), p_i(a^1), \ldots, p_i(a^{n-1})] , \quad (8)$$

where $p_i(x)$ is a polynomial with all elements in the ith row of U as its coefficients, that is, $p_i(x) = \sum_{j=0}^{d-1} u_{ij} x^j$, and a is a generator of $GF(2^m)$. In matrix form, we have

$$U \cdot G = C,$$

where

$$G = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ a^0 & a^1 & \cdots & a^{n-1} \\ (a^0)^2 & (a^1)^2 & \cdots & (a^{n-1})^2 \\ & & \vdots & \\ (a^0)^{d-1} & (a^1)^{d-1} & \cdots & (a^{n-1})^{d-1} \end{bmatrix},$$

and C is the codeword vector with dimension $(\alpha \times n)$. Note that each row of C is decoded separately in all decoding procedures. Finally, the ith column of C is distributed to storage node i for $1 \leq i \leq n$.

The generator matrix G of the RS code can be reformulated as

$$\begin{aligned} G &= \begin{bmatrix} 1 & 1 & \cdots & 1 \\ a^0 & a^1 & \cdots & a^{n-1} \\ (a^0)^2 & (a^1)^2 & \cdots & (a^{n-1})^2 \\ & & \vdots & \\ (a^0)^{\alpha-1} & (a^1)^{\alpha-1} & \cdots & (a^{n-1})^{\alpha-1} \\ (a^0)^\alpha 1 & (a^1)^\alpha 1 & \cdots & (a^{n-1})^\alpha 1 \\ (a^0)^\alpha a^0 & (a^1)^\alpha a^1 & \cdots & (a^{n-1})^\alpha a^{n-1} \\ (a^0)^\alpha (a^0)^2 & (a^1)^\alpha (a^1)^2 & \cdots & (a^{n-1})^\alpha (a^{n-1})^2 \\ & & \vdots & \\ (a^0)^\alpha (a^0)^{\alpha-1} & (a^1)^\alpha (a^1)^{\alpha-1} & \cdots & (a^{n-1})^\alpha (a^{n-1})^{\alpha-1} \end{bmatrix} \\ &= \begin{bmatrix} \bar{G} \\ \bar{G}\Delta \end{bmatrix}, \end{aligned}$$

where $\bar{G}$ contains the first α rows in G and Δ is a diagonal matrix with $(a^0)^\alpha,\ (a^1)^\alpha,\ (a^2)^\alpha, \ldots,\ (a^{n-1})^\alpha$ as diagonal elements. It is easy to see that the α symbols stored in storage node i is

$$U \cdot \begin{bmatrix} \boldsymbol{g}_i^T \\ (a^{i-1})^\alpha \boldsymbol{g}_i^T \end{bmatrix} = A_1 \boldsymbol{g}_i^T + (a^{i-1})^\alpha A_2 \boldsymbol{g}_i^T,$$

where $\boldsymbol{g}_i^T$ is the ith column in $\bar{G}$. Note that the decoding procedure in [10] requires $(a^0)^\alpha,\ (a^1)^\alpha,\ (a^2)^\alpha, \ldots,\ (a^{n-1})^\alpha$ to be all distinct. This can be guaranteed if this code is over $GF(2^m)$ for $m \geq \lceil \log_2 n\alpha \rceil$.

A final remark is that each column in G can be generated by knowing the index of the column and the generator a. Therefore, each storage node does not need to store the entire G to perform exact-regeneration.

C. Decoding for Data Reconstruction

The generator polynomial of the RS code encoded by (8) has $a^{n-d}, a^{n-d-1}, \ldots, a$ as roots [12]. Without loss of generality, we assume that the data collector retrieves encoded symbols from k storage nodes $j_0,\ j_1, \ldots,\ j_{k-1}$. First, the information sequence $\boldsymbol{m}$ is recovered by the procedure given in [10]. If the recovered information sequence does not pass the CRC, then we need to perform the error-erasure decoding. In addition to the received encoded symbols from k storage nodes, the data collector needs to retrieve the encoded symbols from $d+2-k$ storage nodes of the remaining storage nodes. The data collector then performs error-erasure decoding to obtain $\tilde{C}$, the first d columns of the codeword vector. Let $\hat{G}$ be the first d columns of G. Then the recovered information sequence can be obtained from

$$\tilde{U} = \tilde{C} \cdot \hat{G}^{-1}, \quad (9)$$

where $\hat{G}^{-1}$ is the inverse of $\hat{G}$ and it always exists. If the recovered information sequence passes the CRC, it is done; otherwise, two more symbols need to be retrieved. The data collector continues the decoding process until it successfully recovers the correct information sequence or no more storage nodes can be accessed. In each step, the progressive decoding that we proposed in [19] is applied to reduce the computation complexity. Note that the RS code used is capable of correcting up to $\lfloor (n-d)/2 \rfloor$ errors.

The decoding algorithm is summarized in Algorithm 1. Note that, in practice, Algorithm 1 will be repeated β times for each retrieved symbol when $\beta > 1$.

D. Verification for Regeneration

To verify whether the recovered data are the same as those stored in the failed node, integrity check is needed. However, such check should be performed based on information stored on nodes *other than* the failed node. We consider two mechanisms for verification.

In the first scheme, each storage node keeps the CRC checksums for the remaining $n-1$ storage nodes. When the helper accesses d surviving storage nodes, it also asks for the CRC checksums for the failed node from them. Using the majority vote on all received CRC checksums, the helper can

Algorithm 1: Decoding of MSR Codes for Data Reconstruction

```
begin
    The data collector randomly chooses $k$ storage nodes
    and retrieves encoded data, $Y_{\alpha\times k}$;
    Perform the procedure given in [10] to recover $\tilde{\boldsymbol{m}}$;
    if $CRCTest(\tilde{\boldsymbol{m}}) = SUCCESS$ then
        return $\tilde{\boldsymbol{m}}$;
    else
        Retrieve $d-k$ more encoded data from remaining
        storage nodes and merge them into $Y_{\alpha\times d}$;
        $i \leftarrow d$;
        while $i \le n-2$ do
            $i \leftarrow i+2$;
            Retrieve two more encoded data from
            remaining storage nodes and merge them into
            $Y_{\alpha\times i}$;
            Perform progressive error-erasure decoding
            on each row in $Y$ to recover $\tilde{C}$;
            Obtain $\tilde{U}$ by (9) and convert it to $\tilde{\boldsymbol{m}}$;
            if $CRCTest(\tilde{\boldsymbol{m}}) = SUCCESS$ then
                return $\tilde{\boldsymbol{m}}$;
        return FAIL;
```

obtain the correct CRC checksum if no more than $\lfloor(d-1)/2\rfloor$ accessed storage nodes are compromised. To see the storage complexity of this scheme, let us take a numerical example. Consider a [100, 20, 38] MSR code with $\beta = 1000, \alpha = 19\times\beta = 19000, B = 4.18\text{Mb}$. The total number of bits stored in each node is then $19000\times 11 = 209000$. If a 32-bit CRC checksum is added to each storage node, the redundancy is $r(n-1)/\alpha m = 32\times 99/209000 \approx 1.5\%$ and the extra bandwidth for transmitting the CRC checksums is around $rd/\alpha m = 1216/418000 \approx 0.3\%$. Hence, both redundancy for storage and bandwidth are manageable for large β's.

When β is small, we adopt an error-correcting code to encode the r-bit CRC checksum. This can improve the storage and bandwidth efficiency. First we select the operating finite field $GF(2^{m'})$ such that $2^{m'} \ge n-1$. Then an $[n-1, k']$ RS code with $k' = \lceil r/m'\rceil$ is used to encode the CRC checksum. Note that this code is different from the RS code used for MSR data regenerating. In encoding the CRC checksum of a storage node into $n-1$ symbols and distributing them to the $n-1$ other storage nodes, extra $(n-1)m'$ bits are needed on each storage node. When the helper accesses d storage nodes to repair the failed node i, these nodes also send out the symbols associated with the CRC checksum for node i. The helper then can perform error-erasure decoding to recover the CRC checksum. The maximum number of compromised storage nodes among the accessed d nodes that can be handled by this approach is $\lfloor(d-k')/2\rfloor$ and the extra bandwidth is dm'. Since m' is much smaller than $n-1$ and r, the redundancy for storage and bandwidth can be reduced.

E. Decoding for Regeneration

Let node i be the failed node to be recovered. During regeneration, the helper accesses s surviving storage nodes, where $d \le s \le n-1$. Without loss of generality, we assume that the storage nodes accessed are j_0, j_1,..., j_{s-1}. Every accessed node takes the inner product between its α symbols and

$$\boldsymbol{g}_i = [1, (a^{i-1})^1, (a^{i-1})^2, \dots, (a^{i-1})^{\alpha-1}] \ , \tag{10}$$

where $\boldsymbol{g}_i$ can be generated by index i and the generator a, and sends the resultant symbol to the helper. Since the MSR code is a linear code, the resultant symbols transmitted, $y_{j_0}, y_{j_1}, y_{j_2}, \dots, y_{j_{s-1}}$, can be decoded to the codeword $\boldsymbol{c}$, where

$$\begin{aligned}\boldsymbol{c} &= \boldsymbol{g}_i \cdot (U\cdot G)\\ &= (\boldsymbol{g}_i\cdot U)\cdot G \ ,\end{aligned}$$

if $(n-s)+2e < n-d+1$, where e is the number of errors among the s resultant symbols. Multiplying $\boldsymbol{c}$ by the inverse of the first d columns of G, i.e., $\hat{G}^{-1}$, one can recover

$$\boldsymbol{g}_i \cdot U$$

which is equivalent to

$$\boldsymbol{g}_i \cdot [A_1 \ A_2] = [\boldsymbol{g}_i\cdot A_1 \ \boldsymbol{g}_i \cdot A_2] \ .$$

Recall that $\boldsymbol{g}_i$ is the transpose of ith column of $\bar{G}$, the first α rows in G. Since A_j, for $j = 1,2$, are symmetric matrices, $(\boldsymbol{g}_i A_j)^T = A_j \boldsymbol{g}_i^T$. The α symbols stored in the failed node i can then be calculated as

$$(\boldsymbol{g}_i A_1)^T + (a^{i-1})^\alpha (\boldsymbol{g}_i A_2)^T \ . \tag{11}$$

The progressive decoding procedure in [19] can be applied in decoding $y_{j_0}, y_{j_1}, y_{j_2}, \dots, y_{j_{s-1}}$. First, the helper accesses d storage nodes and decodes $y_{j_0}, y_{j_1}, y_{j_2}, \dots, y_{j_{d-1}}$ to obtain $\boldsymbol{c}$ and α symbols by (11). Then, it verifies the CRC checksum. If the CRC test is passed, the regeneration is successful; otherwise, two more surviving storage nodes need to be accessed. Then the helper decodes the received $y_{j_0}, y_{j_1}, y_{j_2}, \dots, y_{j_{d+1}}$ to obtain $\boldsymbol{c}$ and recover α symbols. The process repeats until a sufficient number of correctly stored data have been retrieved to recover the failed node. Again, in practice, when $\beta > 1$, the decoding needs to be performed β times to recover $\beta\alpha$ symbols before verifying the CRC checksum. The data regenerating algorithm is summarized in Algorithm 2.

IV. Encoding and Decoding of Error-Correcting Exact-Regenerating Codes for the MBR Points

In this section we demonstrate that by selecting the same RS codes as that for MSR codes and designing a proper decoding procedure, the MBR codes in [10] can be extended to handle Byzantine failures. Since the verification procedure for MBR codes is the same as that of MSR codes, it is omitted.

Algorithm 2: Decoding of MSR Codes for Regeneration

```
begin
    Assume node $i$ is failed.
    The helper randomly chooses $d$ storage nodes;
    Each chosen storage node combines its symbols as a $(\beta \times \alpha)$ matrix and multiply it by $\boldsymbol{g}_i$ in (10);
    The helper collects these resultant vectors as a $(\beta \times d)$ matrix $Y$.
    The helper obtains the CRC checksum for node $i$;
    $i \leftarrow d$;
    repeat
        Perform progressive error-erasure decoding on each row in $Y$ to recover $\tilde{C}$ (error-erasure decoding performs $\beta$ times);
        $M = \tilde{C}\hat{G}^{-1}$, where $\hat{G}^{-1}$ is the inverse of the first $d$ columns of $G$;
        Obtain the $\beta\alpha$ information symbols, $\boldsymbol{s}$, from $M$ by the method given in (11);
        if $CRCTest(\boldsymbol{s}) = SUCCESS$ then
            return $\boldsymbol{s}$;
        else
            $i \leftarrow i + 2$;
            The helper accesses two more remaining storage nodes;
            Each chosen storage node combines its symbols as a $(\beta \times \alpha)$ matrix and multiply it by $\boldsymbol{g}_i$ given in (10);
            The helper merges the resultant vectors into $Y_{\beta \times i}$;
    until $i \geq n - 2$;
    return FAIL;
```

A. Encoding

Let the information sequence $\boldsymbol{m} = [m_0, m_1, \ldots, m_{B-1}]$ be arranged into an information vector U with size $\alpha \times d$ such that

$$u_{ij} = \begin{cases} u_{ji} = m_{k_1} & \text{for } i \leq j \leq k \\ u_{ji} = m_{k_2} & \text{for } k+1 \leq i \leq d,\ 1 \leq j \leq k \\ 0 & \text{otherwise} \end{cases},$$

where $k_1 = (i-1)(k+1) - i(i+1)/2 + j$ and $k_2 = (i-k-1)k + k(k+1)/2 + j$. In matrix form, we have

$$U = \begin{bmatrix} A_1 & A_2^T \\ A_2 & \mathbf{0} \end{bmatrix}, \tag{12}$$

where A_1 is a $k \times k$ matrix, A_2 a $(d-k) \times k$ matrix, $\mathbf{0}$ is the $(d-k) \times (d-k)$ zero matrix. Both A_1 and A_2 are symmetric. It is clear that U has a dimension $d \times d$ (or $\alpha \times d$).

We apply an $[n, d]$ RS code to encode each row of U. Let $p_i(x)$ be the polynomial with all elements in the ith row of U as its coefficients. That is, $p_i(x) = \sum_{j=0}^{d-1} u_{ij}x^j$. The corresponding codeword of $p_i(x)$ is thus

$$[p_i(a^0 = 1), p_i(a^1), \ldots, p_i(a^{n-1})]. \tag{13}$$

Recall that a is a generator of $GF(2^m)$. In matrix form, we have

$$U \cdot G = C,$$

where

$$G = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ a^0 & a^1 & \cdots & a^{n-1} \\ (a^0)^2 & (a^1)^2 & \cdots & (a^{n-1})^2 \\ & & \vdots & \\ (a^0)^{k-1} & (a^1)^{k-1} & \cdots & (a^{n-1})^{k-1} \\ (a^0)^k & (a^1)^k & \cdots & (a^{n-1})^k \\ & & \vdots & \\ (a^0)^{d-1} & (a^1)^{d-1} & \cdots & (a^{n-1})^{d-1} \end{bmatrix},$$

and C is the codeword vector with dimension $(\alpha \times n)$. G is called the generator matrix of the $[n, d]$ RS code. G can be divided into two sub-matrices as

$$G = \begin{bmatrix} G_k \\ B \end{bmatrix},$$

where

$$G_k = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ a^0 & a^1 & \cdots & a^{n-1} \\ (a^0)^2 & (a^1)^2 & \cdots & (a^{n-1})^2 \\ & & \vdots & \\ (a^0)^{k-1} & (a^1)^{k-1} & \cdots & (a^{n-1})^{k-1} \end{bmatrix} \tag{14}$$

and

$$B = \begin{bmatrix} (a^0)^k & (a^1)^k & \cdots & (a^{n-1})^k \\ & & \vdots & \\ (a^0)^{d-1} & (a^1)^{d-1} & \cdots & (a^{n-1})^{d-1} \end{bmatrix}.$$

Note that G_k is a generator matrix of the $[n, k]$ RS code and it will be used in the decoding process for data reconstruction.

B. Decoding for Data Reconstruction

The generator polynomial of the RS code encoded by (14) has $a^{n-k}, a^{n-k-1}, \ldots, a$ as roots [12]. Hence, the progressive decoding scheme given in [19] can be applied to decode the proposed code if there are errors in the retrieved data. Unlike the decoding procedure given in III-C, where an $[n, d]$ RS decoder is applied, we need an $[n, k]$ RS decoder for MBR codes.

Without loss of generality, we assume that the data collector retrieves encoded symbols from s storage nodes $j_0, j_1, \ldots, j_{s-1}$, $k \leq s \leq n$. Recall that $\alpha = d$ in MBR. Hence, the data collector receives d vectors where each vector has s symbols. Collecting the first k vectors as Y_k and the remaining $d-k$ vectors as Y_{d-k}. From (12), we can view the codewords in the last $d-k$ rows of C as being encoded by G_k instead of G. Hence, the decoding procedure of $[n, k]$ RS codes can be applied on Y_{d-k} to recover the codewords in the last $d-k$ rows of C. Let $\hat{G}_k$ be the first k columns of G_k and $\tilde{C}_{d-k}$ be the recovered codewords in the last $d-k$ rows of C. A_2 in U can be recovered as

$$\tilde{A}_2 = \tilde{C}_{d-k} \cdot \hat{G}_k^{-1}. \tag{15}$$

We then calculate $\tilde{A}_2^T \cdot B$ and only keep the j_0th, j_1th, ..., j_{s-1}th columns of the resultant matrix as E, and subtract E from Y_k:

$$Y_k' = Y_k - E. \tag{16}$$

Applying the RS decoding algorithm again on Y_k' we can recover A_1 as

$$\tilde{A}_1 = \tilde{C}_k \cdot \hat{G_k}^{-1} \ . \tag{17}$$

CRC checksum is computed on the decoded information sequence to verify the recovered data. If the CRC test is passed, the data reconstruction is successful; otherwise the progressive decoding procedure is applied, where two more storage nodes need to be accessed from the remaining storage nodes in each round until no further errors are detected. The data reconstruction algorithm is summarized in Algorithm 3.

Algorithm 3: Decoding of MBR Codes for Data Reconstruction

```
begin
    The data collector randomly chooses $k$ storage nodes
    and retrieves encoded data, $Y_{d\times k}$;
    $i \leftarrow d$;
    repeat
        Perform progressive error-erasure decoding on
        last $d-k$ rows in $Y$ to recover $\tilde{C}$ (error-erasure
        decoding performs $d-k$ times);
        Calculate $\tilde{A}_2$ via (15);
        Calculate $\tilde{A}_2 \cdot B$ and obtain $Y_k'$ via (16);
        Perform progressive error-erasure decoding on $Y_k'$
        to recover the first $k$ rows in codeword vector
        (error-erasure decoding performs $k$ times);
        Calculate $\tilde{A}_1$ via (17);
        Recover the information sequence $\boldsymbol{s}$ from $\tilde{A}_1$ and
        $\tilde{A}_2$;
        if $CRCTest(\boldsymbol{s}) = SUCCESS$ then
            return $\boldsymbol{s}$;
        else
            $i \leftarrow i+2$;
            Retrieve two more encoded data from
            remaining storage nodes and merge them into
            $Y_{d\times i}$;
    until $i \geq n-2$;
    return FAIL;
```

C. Decoding for Regeneration

Decoding for regeneration with MBR is very similar to that with MSR. After obtaining $g_i \cdot U$, we take its transposition. Since U is symmetric, we have $U^T = U$ and

$$U^T \cdot g_i^T = U \cdot g_i^T \ .$$

A CRC test is performed on all $\beta\alpha$ symbols. If the CRC test is passed, the $\beta\alpha$ symbols are the data stored in the failed node; otherwise, the progressive decoding procedure is applied.

V. Analysis

In this section, we provide an analytical study of the fault-tolerant capability, security strength, and storage and bandwidth efficiency of the proposed schemes.

A. Fault-tolerant capability

In analyzing the fault-tolerant capability, we consider two types of failures, namely crash-stop failures and Byzantine failures. Nodes are assumed to fail independently (as opposed in a coordinated fashion). In both cases, the fault-tolerant capacity is measured by the maximum number of failures that the system can handle to remain functional.

Crash-stop failure: Crash-stop failures can be viewed as erasure in the codeword. Since at least k nodes need to be available for data reconstruction, it is easy to show that the maximum number of crash-stop failures that can be tolerated in data reconstruction is $n-k$. For regeneration, d nodes need to be accessed. Thus, the fault-tolerant capability is $n-d$. Note that since live nodes all contain correct data, the associated CRC checksums are also correct.

Byzantine failure: In general, in RS codes, two additional correct code fragments are needed to correct one erroneous code fragments. However, in the case of data regeneration, the capability of the helper to obtain the correct CRC checksum also matters. In the analysis, we assume that the error-correction code is used in the process to obtain the correct CRC checksum. Data regeneration will fail if the helper cannot obtain the correct CRC checksum even when the number of failed nodes is less than the maximum number of faults the RS code can handle. Hence, we must take the minimum of the capability of the RS code (in MBR and MSR) and the capability to recover the correct CRC checksum. Thus, with MSR and MBR code, $\lfloor \frac{n-d}{2} \rfloor$ and $\lfloor \frac{n-k}{2} \rfloor$ erroneous nodes can be tolerated in data reconstruction. On the other hand, the fault-tolerant capacity of MSR and MBR code for data regeneration are both $\min\left\{\lfloor \frac{n-d}{2} \rfloor, \lfloor \frac{d-k'}{2} \rfloor\right\}$.

B. Security Strength

In analyzing the security strength, we consider forgery attacks, where polluters [9], a type of Byzantine attackers, try to disrupt the data reconstruction and regenerating process by forging data cooperatively. In other words, collusion among polluters are considered. We want to determine the minimum number of polluters to forge the data in data reconstruction and regeneration. The security strength is therefore one less the number. Forgery in data regeneration is useful when an attacker only has access to a small set of nodes but through the data regeneration process "pollutes" the data on other storage nodes and thus ultimately leads to valid but erroneous data reconstruction.

In data reconstruction, for worst case analysis, we consider the security strength such that only one row of U is modified.[3] Let the polluters be $j_0, j_1, \ldots, j_{v-1}$, who can collude to forge the information symbols. Suppose that $\boldsymbol{y}$ is the forged row in U. Let $\tilde{\boldsymbol{y}} = \boldsymbol{y} + \boldsymbol{u}$, where $\boldsymbol{u}$ is the real information symbols in the row of U. Then, according to the RS encoding procedure, we have

$$\boldsymbol{y}G = (\tilde{\boldsymbol{y}} + \boldsymbol{u})G = \tilde{\boldsymbol{y}}G + \boldsymbol{u}G = \boldsymbol{v} + \boldsymbol{c}, \tag{18}$$

[3]Due to symmetry in U, most of the time, making changes on a row in U results in changes on several rows simultaneously.

TABLE I
EVALUATION OF MSR AND MBR CODES

	MSR code		MBR code	
	Data reconstruction	Regeneration	Data reconstruction	Regeneration
Fault-tolerant capability against erasures	$n-k$	$n-d$	$n-k$	$n-d$
Fault-tolerant capacity against Byzantine faults	$\lfloor\frac{n-d}{2}\rfloor$	$\min\{\lfloor\frac{n-d}{2}\rfloor, \lfloor\frac{d-k'}{2}\rfloor\}$	$\lfloor\frac{n-k}{2}\rfloor$	$\min\{\lfloor\frac{n-d}{2}\rfloor, \lfloor\frac{d-k'}{2}\rfloor\}$
Security strength under forgery attack	$\min\{k, \lceil\frac{n-d+2}{2}\rceil\}-1$	$\min\{d, \lceil\frac{n-d+2}{2}\rceil\}-1$	$\min\{k, \lceil\frac{n-k+2}{2}\rceil\}-1$	$\min\{d, \lceil\frac{n-d+2}{2}\rceil\}-1$
Redundancy ratio on storage (bits)	$\frac{r}{mk\alpha-r}$	$\frac{(n-1)m'}{\beta\alpha m}$	$\frac{r}{m(kd-k(k-1)/2)-r}$	$\frac{(n-1)m'}{\beta\alpha m}$
Redundancy ratio on bandwidth (bits)	$\cdot$	$\frac{dm'}{\beta md}=\frac{m'}{\beta m}$	$\cdot$	$\frac{dm'}{\beta md}=\frac{m'}{\beta m}$

where $k'=\lfloor\frac{r}{m'}\rfloor$ and $m'=\lceil\log_2(n-1)\rceil$

where $\boldsymbol{c}$ is the original data storage in storage nodes and $\boldsymbol{v}$ is the modified data must be made by the polluters. Let the number of non-zero symbols in $\boldsymbol{v}$ is h. It is clear that $h \geq n-d+1$, where $n-d+1$ is the minimum Hamming distance of the RS code, since $\boldsymbol{v}$ must be a codeword. For worst-case consideration, we assume that $h = n-d+1$. In order to successfully forge information symbols, the attacker must compromise some storage nodes and make them to store the corresponding encoded symbols in $\boldsymbol{y}G$, the codeword corresponding to the forged information symbols. If the attacker compromises k storage nodes, then when the data collector happens to access these compromised storage nodes, according to the decoding procedure, the attack can forge the data successfully. Let the attacker compromise $b < k$ storage nodes. According the decoding procedure, when $h-b = n-d+1-b \leq \lfloor\frac{n-d}{2}\rfloor$, where $\lfloor\frac{n-d}{2}\rfloor$ is the error-correction capability of the RS code, the decoding algorithm still has chance to decode the received vector to $\boldsymbol{y}G$.[4] Taking the smallest value of b we have $b = \lceil\frac{n-d+2}{2}\rceil$. Hence, the security strength for data reconstruction is $\min\{k, \lceil\frac{n-d+2}{2}\rceil\}-1$ in MSR codes. Since the $[n,k]$ RS code is used in decoding for MBR codes, the security strength for them becomes $\min\{k, \lceil\frac{n-k+2}{2}\rceil\}-1$.

Next we investigate the forgery attack on regeneration. Since computing the CRC checksum is a linear operation, there is no need for the attacker to break the CRC checksum for the failed node. It only needs to make the forged data with all zero redundant bits. Hence, the security strength for regeneration is $\min\{d, \lceil\frac{n-d+2}{2}\rceil\}-1$.

It can be observed that CRC does not increase the security strength in forgery attack. By using hash value, the security strength can be increased since the operation to obtain hash value is non-linear. In this case, the attacker not only needs to obtain the original information data but also can forge hash value. Hence, the security strength can be increased to at least $k-1$ in data reconstruction and at least $d-1$ for regeneration.[5]

C. Redundancy Ratios on Storage and Bandwidth

CRC checksums incur additional overhead in storage and bandwidth consumption. The amount of redundancy incurred for data construction is r bits, i.e., the size of the CRC checksum. Each information sequence is appended with the extra r bits such that it can be verified after reconstruction. The number of information bits is $mk\alpha - r$ for MSR codes and $m(kd-k(k-1)/2)-r$ for MBR codes, respectively. For regeneration, we assume that the $[n-1, k']$ RS code is used to distribute the encoded CRC symbols to $n-1$ storage nodes, where $k' = \lfloor\frac{r}{m'}\rfloor$ and $m' = \lceil\log_2(n-1)\rceil$. Since each storage node must store the encoded CRC symbols for other $n-1$ storage nodes, the extra storage required for it is $(n-1)m'$ bits. The encoded data symbols stored in each storage node is $\beta\alpha m$ bits.

The helper must obtain the correct CRC checksum for the failed node to verify the correctness of the recovered data. The d storage nodes accessed need to provide their stored data associated with the CRC checksum of the failed node to the helper. Since each piece has m' bits, the total extra bandwidth is dm'. The total bandwidth to repair the $\beta\alpha$ symbols stored in the failed node is βmd.

Table I summarizes the quantitative results of fault-tolerant capability, security strength, and redundancy ratio of the MSR and MBR codes.

VI. RELATED WORK

Regenerating codes were introduced in the pioneer works by Dimakis *et al.* in [1], [2]. In these works, the so-called cut-set bound was derived which is the fundamental limit for designing regenerating codes. In these works, the data reconstruction and regeneration problems were formulated as a multicast network coding problem. From the cut-set bounds between the source and the destination, the parameters of the regenerating codes were shown to satisfy (1), which reveals the trade-off between storage and repair bandwidth. Those parameters satisfying the cut-set bound with equality were also derived.

The regeneration codes with parameters satisfying the cut-set bound with equality were proposed in [3], [4]. In [3] a deterministic construction of the regenerating codes with $d = n-1$ was presented. In [4], the network coding approach was adopted to design the regenerating codes. Both constructions achieved functional regeneration but not exact regeneration.

Exact regeneration was considered in [5]–[7]. In [5], a search algorithm was proposed to search for exact-regenerating MSR codes with $d = n-1$; however, no systematic construction method was provided. In [6], the MSR codes with

[4]With higher number of compromised nodes, the chance to forge the data increases. In this case, even though data can be decoded successfully, the resulting data differs from the original data.

[5]For regeneration, the security strength is $\max\{d, \min\{k', \lceil\frac{d-k'+2}{2}\rceil\}\}-1 = d-1$ since k' is usually less than d.

$k = 2, d = n-1$ were constructed by using the concept of interference alignment, which was borrowed from the context of wireless communications. A drawback of this approach is that it operates on a finite field with a large size. In [7], the authors provided an explicit method to construct the MBR codes with $d = n-1$. No computation is required for these codes during the regeneration of a failed node. Explicit construction of the MSR codes with $d = k+1$ was also provided; however, these codes can perform exact regeneration only for a subset of failed storage nodes.

In [20], the authors proved that exact regeneration is impossible for MSR codes with $[n, k, d < 2k-3]$ when $\beta = 1$. Based on interference alignment approach, a code construction was provided for the MSR codes with $[n = d+1, k, d \geq 2k-1]$. In [10], the explicit constructions for optimal MSR codes with $[n, k, d \geq 2k-2]$ and optimal MBR codes were proposed. The construction was based on the product of two matrices: information matrix and encoding matrix. The information matrix (or its sub-matrices) is symmetric in order to have exact-regeneration property.

The problem of security on regenerating codes were considered in [8], [9]. In [8], the authors considered the security problem against eavesdropping and adversarial attackers during the regeneration process. They derived upper bounds on the maximum amount of information that can be stored safely. An explicit code construction was given for $d = n-1$ in the bandwidth-limited regime. The problem of Byzantine fault tolerance for regenerating codes was considered in [9]. The authors studied the resilience of regenerating codes which support multi-repairs. By using collaboration among newcomers (helpers), upper bounds on the resilience capacity of regenerating codes were derived. Even though our work also deals with the Byzantine failures, it does not need to have multiple helpers to recover the failures.

The progressive decoding technology for distributed storage was first introduced in [19]. The scheme retrieved just enough data from surviving storage nodes to recover the original data in the presence of crash-stop and Byzantine failures. The decoding was performs incrementally such that both communication and computation cost are minimized.

VII. Conclusions

In this paper, we considered the problem of exact regeneration with error correction capability for Byzantine fault tolerance in distributed storage networks. We showed the Reed-Solomon codes combined with CRC checksums can be used for both data reconstruction and regenerating, realizing MSR and MBR in the later case. Progressive decoding can be applied in both applications to reduce the computation complexity in presence of erroneous data. Analysis on the fault tolerance, security, storage and bandwidth overhead shows that the proposed schemes are effective without incurring too much overhead.

References

[1] A. G. Dimakis, P. B. Godfrey, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," in *Proc. of 26th IEEE International Conference on Computer Communications (INFOCOM)*, Anchorage, Alaska, May 2007, pp. 2000–2008.

[2] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inform. Theory*, vol. 56, pp. 4539 – 4551, September 2010.

[3] Y. Wu, A. G. Dimakis, and K. Ramchandran, "Deterministic regenerating codes for distributed storage," in *Proc. of 45th Annual Allerton Conference on Control, Computing, and Communication*, Urbana-Champaign, Illinois, September 2007.

[4] Y. Wu, "Existence and construction of capacity-achieving network codes for distributed storage," *IEEE Journal on Selected Areas in Communications*, vol. 28, pp. 277 – 288, February 2010.

[5] D. F. Cullina, "Searching for minimum storage regenerating codes," California Institute of Technology Senior Thesis, 2009.

[6] Y. Wu and A. G. Dimakis, "Reducing repair traffic for erasure coding-based storage via interference alignment," in *Proc. IEEE International Symposium on Information Theory*, Seoul, Korea, July 2009, pp. 2276–2280.

[7] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, "Explicit construction of optimal exact regenerating codes for distributed storage," in *Proc. of 47th Annual Allerton Conference on Control, Computing, and Communication*, Urbana-Champaign, Illinois, September 2009, pp. 1243–1249.

[8] S. Pawar, S. E. Rouayheb, and K. Ramchandran, "Securing dynamic distributed storage systems against eavesdropping and adversarial attacks," arXiv:1009.2556v2 [cs.IT] 27 Apr 2011.

[9] F. Oggier and A. Datta, "Byzantine fault tolerance of regenerating codes," arXiv:1106.2275v1 [cs.DC] 12 Jun 2011.

[10] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction," *IEEE Trans. Inform. Theory*, vol. 57, pp. 5227–5239, August 2011.

[11] I. S. Reed and G. Solomon, "Polynomial codes over certain finite field," *J. Soc. Indust. and Appl. Math.(SIAM)*, vol. 8 (2), pp. 300 – 304, 1960.

[12] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

[13] H. William, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C: The art of scientific computing*. Cambridge university press New York, NY, USA, 1988.

[14] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," ICSI Technical Report TR-95-048, 1995.

[15] I. S. Reed and X. Chen, *Error-Control Coding for Data Networks*. Boston, MA: Kluwer Academic, 1999.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[17] R. C. Merkle, "A certified digital signature," in *Proc. of the CRYPTO '89*, Santa Barbara, CA, August 20-24 1989, pp. 218–238.

[18] I. Damgård, "A design principle for hash functions," in *Proc. of the CRYPTO '89*, Santa Barbara, CA, August 20-24 1989, pp. 416–427.

[19] Y. S. Han, S. Omiwade, and R. Zheng, "Survivable distributed storage with progressive decoding," in *Proc. of the 29th Conference of the IEEE Communications Society (Infocom '10)(Mini-conference)*, San Diego, CA, March 15-19 2010.

[20] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Interference alignment in regenerating codes for distributed storage: Necessity and code constructions," arXiv:1005.1634v2[cs.IT] 13 Sep 2010.