An XOR-Based Erasure-Resilient Coding Scheme

Johannes Blömer[*], Malik Kalfane[†], Richard Karp[‡], Marek Karpinski[§]
Michael Luby[¶], David Zuckerman[||]

Abstract

An (m, n, b, r)-erasure-resilient coding scheme consists of an encoding algorithm and a decoding algorithm with the following properties. The encoding algorithm produces a set of n packets each containing b bits from a message of m packets containing b bits. The decoding algorithm is able to recover the message from any set of r packets. Erasure-resilient codes have been used to protect real-time traffic sent through packet based networks against packet losses. In this paper we describe a erasure-resilient coding scheme that is based on a version of Reed-Solomon codes and which has the property that $r = m$. Both the encoding and decoding algorithms run in quadratic time and have been customized to give the first real-time implementations of *Priority Encoding Transmission* (PET) [2],[1] for medium quality video transmission on Sun SPARCstation 20 workstations.

1 Introduction

Most existing and proposed networks are packet based, where a packet is a fixed length indivisible unit of information that either arrives intact upon transmission or is completely lost. This model accurately reflects properties of Internet and ATM-based networks, where local error-correcting codes can be used (and often are used) on individual packets to protect against possible errors as the packet traverses the network. However, the timely arrival of individual packets sent long distances over a variety of heterogeneous networks is a global property that seems to be harder to control on a local basis. Thus, it makes sense to protect real-time traffic sent through such networks against losses by adding a moderate level of redundancy using erasure-resilient codes.

Algorithms based on this approach have been developed for applications such as multicasting real-time high-volume video information over lossy packet based networks [2, 1, 4]

[*]International Computer Science Institute, Berkeley, California. Research supported in part by ESPRIT BR Grant EC-US 030 and by Deutsche Forschungsgemeinschaft (DFG), Habilitationsstipendium Bl 314/4-1

[†]Computer Science Division, University of California at Berkeley and International computer Science Institute, Berkeley, California

[‡]Computer Science Division, University of Californai at Berkeley

[§]Department of Computer Science, University of Bonn and International Computer Science Institute, berkeley, California

[¶]International Computer Science Institute, berkeley, California and Computer Science Division, University of California at Berkeley. Research supported in part by National Science Foundation operating grants CCR-9304722 and NCR-9416101, United States-Israel Binational Science Foundation grant No. 92-00226, and ESPRIT BR Grant EC-US 030.

[||]Department of Computer Science, University of Texas, Austin and other high volume real-time applications [8]. The two most important properties of erasure-resilient codes in these applications are the running times of the encoding and decoding algorithms and the amount of encoding sufficient to recover the message. An erasure-resilient code where any portion of the encoding equal to the length of the message is sufficient to recover the message is called a *maximal distance separable* (MDS) code in the literature (see for example [6]). An ideal erasure-resilient code would be a linear time MDS code, but so far no such code is known.

Theoretically the most efficient MDS codes can be constructed based on evaluating and interpolating polynomials over specially chosen finite fields using Discrete Fourier Transform. Up to logarithmic factors these codes achieve a linear running time. However, they don't perform well in practice and are not competitive in practice with simpler, quadratic time methods except for extremely large messages.

In this paper we show how to customize a version of Reed-Solomon codes so that it yields a (quadratic time) MDS code that runs in real-time for medium quality video transmission on existing workstations [2, 1], i.e., at the rate of a few megabits per second.

The version of Reed-Solomon codes we use is based on Cauchy matrices over finite fields (see for example [6] and [11]). Every square submatrices of a Cauchy matrix is invertible. Therefore Cauchy matrices can be used to implement linear, systematic codes. i.e., the encoding is a linear function of the message and the unencoded message is part of the encoding. Since typically the encoding size is only a constant multiple (between 0.1 and 5) of the message size the last property helps implementing an efficient encoding procedure for these codes.

On the decoding side we achieve running time that decreases with the amount of the unencoded message received. Since often the encoding is only moderately larger than the message itself, this is one of the reasons the decoding procedure runs in real time for bit rates of up to a few megabits.

However, the main gain compared to the method based on the evaluation and interpolation of polynomials is achieved by using a well-known representation of elements in finite fields of the form GF[2^L] by ($L \times L$)-matrices over GF[2] (see [9]). This allows us to replace arithmetic operations on field elements by XOR's of computer words and in practice XOR's of computer words are much more efficient than multiplications in finite fields.

The paper is organized as follows. In Section 2 we introduce the basic terminology used throughout the rest of the paper. In Section 3 we present of more detailed outline of the XOR-code. Section 4 describes the matrix representation elements in finite fields. In Section 5 we state and partially prove the main properties of Cauchy matrices. In Section 6 we combine Cauchy matrices with the matrix representation of finite field elements to obtain the XOR-code. We also analyze the running times of the encoding and decoding algorithms. Section 7 contains information on the actual implementation of these algorithms and we present some timing information for these implementations. In Section 8 we prove a lower bound on the packet size needed by MDS codes. We show that the XOR code achieves a nearly optimal packet size. Although the packet size is almost of no concern if Internet packets (typically 1000 bytes) are used, it is a more serious constraint for ATM packets (48 bytes). The lower bound we prove applies only to the payload of packets. It does not include the space needed for the unique identifier, we assume is included in every packet.

Finally, we should mention that recently [3] constructed erasure-resilient codes that are almost MDS and that have linear time encoding and decoding algorithms. In these codes the number of packets needed to recover the message is $(1+\epsilon)m$, where ϵ is an arbitrary positive constant. It yet needs to be determined whether these codes perform well in practice.

It remains an interesting open question to design an MDS code with linear time encoding and decoding algorithms.

2 Terminology

For our applications we utilize the following definitions of erasure-resilient codes and MDS codes.

Definition 2.1 *An erasure-resilient code, specified by a quadruple (m, n, b, r), is a function E that maps messages $M = (M_1, \ldots, M_m)$ of m packets each of size b onto encodings $E(M) = (E_1(M), \ldots, E_n(M))$ of n packets each of size b, such that any r packets $E_{i_1}(M), \ldots, E_{i_r}(M)$ of $E(M)$ together with the indices i_j uniquely determine the message M. Here packets of size b are bit strings of length b. The code is said to be maximum distance separable (MDS) iff $r = m$.*

It is easy to see that a code is MDS if the encoding of two different messages differs in at least $n - m + 1$ packets. Hence our definition coincides with the standard definition of MDS codes (see for example [6]).

As opposed to error-correcting codes, where bits can be corrupted and the locations of corrupted bits are not known in advance, in an erasure-resilient code the indices of corrupted packets are known. The corrupted packets are treated as being lost. In applications of erasure-resilient codes like robust data transfer on packet-based networks [1], information dispersal [8], or secret sharing [10], this is a realistic assumption.

An important subclass of error-correcting or erasure-resilient codes are *linear codes*. For a linear erasure-resilient code over the field GF$[2^L]$ the b bits of a packet are considered as describing b/L elements in the finite field GF$[2^L]$. A message is viewed as an element of the vector space $(\mathrm{GF}[2^L])^{mb/L}$. A code is called linear, if the function E is linear. Hence the code can be described by a $(nb/L \times mb/L)$-matrix over GF$[2^L]$. This matrix is called the *generator matrix* of the code.

An erasure-resilient code is *systematic*, if the first m packets of the encoding of a message M are the packets of the message M itself. The first m packets of the encoding will be called *information packets*. The remaining $n - m$ encoding packets are called *redundant packets*.

A linear code over GF$[2^L]$ is a systematic code if the first mb/L rows of its generator matrix form the identity matrix. For two matrices A, B, where A is a $(k \times m)$-matrix and B is a $(l \times m)$-matrix denote by $(A|B)$ the $(K + l) \times m)$-matrix whose first k rows are the rows of A and the last l rows are the rows of B. Let I_m denote the $(m \times m)$-identity matrix over an arbitrary field. A proof of the following theorem can be found for example in [6].

Theorem 2.2 *Let C be a an $(n - m \times m)$-matrix over* GF$[2^L]$. *The matrix $(I_m|C)$ is the generator matrix of a systematic MDS code with packet size L if and only if every square submatrix of C is invertible.*

The main goal of this paper is to describe a systematic linear MDS code over GF[2] such that the time needed to decode a message M from m given encoding packets decreases with the number of information packets among these encoding packets. We first give an outline of the construction.

3 Outline of the XOR-Code

The goal of this paper is to construct a systematic, linear erasure-resilient MDS code over GF[2]. As it turns out there is a general method to turn any systematic, linear code over a finite field GF[2^L] with packet size b into a systematic, linear code over GF[2] with packet size b.

This method is based on the fact that for each element of a finite field GF[2^L] there is a representation as a column vector of length L over GF[2] and a representation as an $(L \times L)$-matrix over GF[2] such that the matrix-vector multiplication of the matrix representation of an element α with the vector representation of an element β yields the vector representation of the product $\alpha\beta$. Moreover, the addition of the vector or matrix representations of two elements α and β results in the vector or matrix representation of the sum $\alpha + \beta$.

This suggests the following method to transform a linear code over GF[2^L] into a linear code over GF[2]. Replace each element in the generator matrix by its matrix representation to obtain the new generator matrix. Replace each field element in a message by its vector representation. It is not hard to see using the above mentioned facts that the encoding of a message using the new code is exactly the encoding using the original code with each field element in the encoding replaced by its vector representation.

The systematic, linear codes to which we apply this general method are based on Cauchy matrices. As required by Theorem 2.2, every square submatrix of a Cauchy matrix is nonsingular. Cauchy matrices are easier to invert than general matrices. This is important to achieve efficient decoding algorithms. MDS codes based on Cauchy matrices are a variant of Reed-Solomon codes (see for example [11]). We will describe Cauchy matrices in detail in Section 5.

In practice it turns out to be more efficient to define packets as containing b words consisting of w bits each than to define them in terms of single bits. In our implementations we chose w to be 32, the word size on a Sun SPARCstation. For a code over GF[2], a message of m packets is then considered to be an $(mb \times 32)$-matrix M over GF[2]. Using the same generator matrix as in the single bit case, the encoding is now given by the matrix product of the generator matrix with M.

Using this slight variation of the XOR-code, encoding a message of size $32m \cdot L$ bits requires $m \cdot L$ coordinate-wise XOR's of 32-bit words. In the original scheme encoding a message of the same length requires $32m \cdot L$ XOR's of single bits. This yields a significant improvement, since the XOR of a computer word of size 32 turns out as fast as the XOR of single bits.

In practice, packets can often be very large. For example, IP packets contain up to 2000 bytes. In this case, it is more efficient to further split the packets into segments of size $32 \cdot L$ rather than increasing the parameter L. We then treat the i^{th} segments in each message packet as the packets of a single message with the parameters as described above. To this partial message we apply the XOR-code and put its encoding into the i^{th} segments of the encoding packets. In other words, if the number of segments in each packet is N we apply the code in parallel to N messages.

More formally, a message M consisting of m packets of size $32 \cdot L \cdot N$ is considered as a $(mL \times 32N)$-matrix over GF[2]. The encoding is obtained by multiplying this matrix by the generator matrix defining the original XOR-code. The j^{th} packet of the encoding is given by the j^{th} block of L consecutive rows of the resulting matrix.

In this version the XOR-code has been implemented. A more detailed description of the implementation and some experimental results will be described in Section 7. In the next sections we will describe and prove the correctness of the XOR-code, for the case where the number of segments in a packet is 1.

4 The matrix representation of finite fields

Let $p(X)$ be an irreducible polynomial of degree L in $\mathrm{GF}[2][X]$. The field $\mathrm{GF}[2^L]$ is isomorphic to $\mathrm{GF}[2][X]/(p(X))$, the field of polynomials in $\mathrm{GF}[2][X]$ taken modulo $p(X)$. Elements in $\mathrm{GF}[2^L]$ can be identified with polynomials $f(X) = \sum_{i=0}^{L-1} f_i X^i$, $f_i \in \mathrm{GF}[2]$, of degree at most $L-1$. The column vector $(f_0, \ldots, f_{L-1})^t$ in $\mathrm{GF}[2]^L$ will be called the *coefficient vector* of the element $f(X) = \sum_{i=0}^{L-1} f_i X^i$.

Construction 4.1 (Matrix representation of finite fields) *For any $f \in \mathrm{GF}[2^L]$ let $\tau(f)$ be the matrix whose i^{th} column is the coefficient vector of $X^{i-1}f \mod p(X)$.*

Lemma 4.2 *τ is a field isomorphism from $\mathrm{GF}[2^L]$ to $\tau(\mathrm{GF}[2^L])$. In particular,*

(i) $\tau(0)$ is the all-zero-matrix.

(ii) $\tau(1)$ is the identity matrix.

(iii) τ is injective.

(iv) For any two field elements f, g, $\tau(f+g) = \tau(f) + \tau(g)$.

(v) For any two field elements f, g, $\tau(fg) = \tau(f)\tau(g)$.

Proof: It suffices to prove $(i) - (v)$. (i), (ii) and (iii) are obvious. (iv) follows from

$$X^i(f+g) \equiv X^i f + X^i g \mod p(X).$$

To prove (v) denote by $f^{(i)}$ the i^{th} column of $\tau(f)$. Also let $\left(g_0^{(j)}, \ldots, g_{L-1}^{(j)}\right)^t$ be the j^{th} column of $\tau(g)$, hence

$$\sum_{i=0}^{L-1} g_i^{(j)} X^i \equiv X^{j-1} g \mod p(X).$$

The j^{th} column of $\tau(f)\tau(g)$ is $\sum_{i=0}^{L-1} g_i^{(j)} f_i$. This is the coefficient vector of

$$\sum_{i=0}^{L-1} g_i^{(j)} (X^{i-1} f) \equiv f \sum_{i=0}^{L-1} g_i^{(j)} X^{i-1} \mod p(X).$$

$\sum_{i=0}^{L-1} g_i^{(j)} X^{i-1} \equiv X^{j-1} g \mod p(X)$, hence the j^{th} column of $\tau(f)\tau(g)$ is the coefficient vector of $X^{j-1}fg$, which is the j^{th} column of $\tau(fg)$. $\square$

Observe that the definition of τ is constructive. $\tau(f)$ can easily be computed using polynomial multiplication and division with remainder. However, in the implementation of the XOR-code we will store all coefficient vectors of field elements in a table and use table look-ups to compute τ. Details of the table will be given in a later section.

5 The main properties Cauchy matrices

In this section we describe the main properties of Cauchy matrices. Cauchy matrices have also been used in [8],[5]. They can be used to define one variant of Reed-Solomon codes [11]).

Definition 5.1 *Let F be a field and let $\{x_1, \ldots, x_m\}$, $\{y_1, \ldots, y_n\}$ be two sets of elements in F such that*

(i) $\forall i \in \{1, \ldots, m\} \, \forall j \in \{1, \ldots, n\} : \; x_i + y_j \neq 0.$

(ii) $\forall i, j \in \{1, \ldots, m\}, i \neq j : \; x_i \neq x_j$ *and* $\forall i, j \in \{1, \ldots, n\}, i \neq j : \; y_i \neq y_j.$

The matrix

$$\begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_n} \\ & & \ddots & \\ \frac{1}{x_{m-1}+y_1} & \frac{1}{x_{m-1}+y_2} & \cdots & \frac{1}{x_{m-1}+y_n} \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \cdots & \frac{1}{x_m+y_n} \end{bmatrix}$$

is called a Cauchy matrix over F.

Theorem 5.2 *Let C be a Cauchy matrix. Every square sub-matrix of C is nonsingular. If*

$$C = \begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_n} \\ & & \ddots & \\ \frac{1}{x_{n-1}+y_1} & \frac{1}{x_{n-1}+y_2} & \cdots & \frac{1}{x_{n-1}+y_n} \\ \frac{1}{x_n+y_1} & \frac{1}{x_n+y_2} & \cdots & \frac{1}{x_n+y_n} \end{bmatrix}$$

then

$$\det(C) = \frac{\prod_{i<j}(x_i - x_j) \prod_{i<j}(y_i - y_j)}{\prod_{i,j=1}^{n}(x_i + y_j)}.$$

A proof of this theorem can be found in [7]. Note that the first part of the theorem follows from the second part, since any sub-matrix of a Cauchy matrix is itself a Cauchy matrix.

Theorem 2.2 and Theorem 5.2 imply that if C is an $(n - m \times m)$-Cauchy matrix over GF$[2^L]$ then $(I_m|C)$ is the generator matrix of a systematic code with packet size L.

Over the finite field GF$[2^L]$ a $(2^{L-1} \times 2^{L-1})$-Cauchy matrix can be constructed as follows. As a set GF$[2^L]$ can be identified with the set of all binary strings of length L. The addition of two elements is the component-wise XOR of the corresponding bit-strings. For each $i = 1, \ldots, 2^{L-1}$, let x_i be the element whose binary expansion is the binary expansion of the integer $i - 1$ and let y_i be the field element whose binary expansion corresponds to the binary expansion of $2^{L-1} + i - 1$. Properties (i) and (ii) of Definition 5.1 are easily verified.

The following theorem can also be found in Rabin's paper [8].

Theorem 5.3 *The inverse of an $(n \times n)$-Cauchy matrix over a field F can be computed using $\mathcal{O}(n^2)$ arithmetic operations in F.*

Proof: Assume the Cauchy matrix C is as in Theorem 5.2. Let $C^{-1} = (d_{ij})$, $i, j = 1, \ldots, n$. The entry d_{ij} is given by

$$d_{ij} = (-1)^{i+j} \frac{\det(C_{ji})}{\det(C)},$$

where C_{ji} is obtained from C by deleting the j^{th} row and the i^{th} column. For each $k = 1, \ldots, n$, let

$$a_k = \prod_{i<k} (x_i - x_k) \prod_{k<j} (x_k - x_j),$$

$$b_k = \prod_{i<k} (y_i - y_k) \prod_{k<j} (y_j - y_k),$$

$$e_k = \prod_{i=1}^{n} (x_k + y_i),$$

and

$$f_k = \prod_{i=1}^{n} (y_k + x_i).$$

By Theorem 5.2

$$\det(C) = \frac{\prod_{k=1}^{n} a_k b_k}{\prod_{k=1}^{n} e_k f_k}$$

and

$$\det(C_{ji}) = \frac{\det(C) e_j f_i}{a_j b_i (x_j + y_i)}.$$

Hence

$$d_{ij} = (-1)^{i+j} \frac{e_j f_i}{a_j b_i (x_j + y_i)}.$$

The $4n$ quantities $a_k, b_k, e_k, f_k, k = 1, \ldots, n$, can be computed using $\mathcal{O}(n^2)$ arithmetic operations. Given these quantities, each d_{ij} can be computed using a constant number of field operations. This proves the theorem. $\Box$

This theorem shows that inverting Cauchy matrices is significantly simpler than inverting an arbitrary matrix. However, multiplications and divisions in a finite field GF[2^L] using the arithmetic of polynomials over GF[2] is rather inefficient. In the implementation of the XOR-code we avoid polynomial arithmetic by transforming multiplications and divisions into additions and subtractions of exponents by using a table of discrete logarithms. Details of the table of logarithms and the matrix inversion algorithm will be given in a later section.

6 An XOR-based MDS code

In this section we construct the XOR-code and analyze its encoding and decoding time. A description of some of the implementation details will be given in a later section.

We will describe the code for messages that consist of m packets each containing of L words of size w. The parameter w can be chosen arbitrarily.

Construction 6.1 (**XOR-Code**) *Assume $L \geq \max\{\log(m), \log(n-m)\}$. Consider a message $M = (M_1, \ldots, M_m)^t$ of m packets containing L words of size w as an element of $(\mathrm{GF}[2])^{mL\times w}$ Let C be an $(n-m \times m)$-Cauchy-matrix over the finite field $\mathrm{GF}[2^L]$. Let $(c_{ij}), i = 1, \ldots, n, j = 1, \ldots, m$, be the matrix $(I_m|C)$. The generator matrix E of the* XOR*-code is given by*

$$E = (\tau(c_{ij})), i = 1, \ldots, n, j = 1, \ldots, m.$$

The j^{th} packet E_j of the encoding of M consists of the rows $r_{jL+1}, \ldots, r_{(j+1)L}$ of the matrix product $E \cdot M$.

Theorem 6.2 *The* XOR*-code is an MDS code.*

Proof: Assume that m packets of the encoding $E \cdot M = (E_1, \ldots, E_n)^t$ are given. Let $I \subseteq \{1, \ldots, m\}$ be the set of indices of the information packets among these m packets, $\overline{I} = \{1, \ldots, m\} \backslash I$. $J \subseteq \{m+1, \ldots, n\}$ is the set of indices of the redundant packets that are given. Hence $|J| = |\overline{I}|$ and $|I| + |J| = m$.

We need to show that the matrix $D = (\tau(c_{ji})), j \in J, i \in \overline{I}$ is invertible. By assumption the matrix $(c_{ji}), j \in J, i \in \overline{I}$ is invertible. It follows from Lemma 4.2 that the inverse of D is given by replacing each entry of the inverse of (c_{ji}) by its matrix representation. $\Box$

Theorem 6.3 *The encoding for the* XOR*-code can be done using $\mathcal{O}(m(n-m)L^2))$* XOR*'s of words of size w.*

The decoding is also not very difficult.

Theorem 6.4 *The decoding for the* XOR*-code can be done using $\mathcal{O}(mkL^2)$* XOR*'s of words of size w and $\mathcal{O}(k^2)$ arithmetic operations in the field* $\mathrm{GF}[2^L]$, *assuming that $m-k$ information packets and k redundant packets are given.*

Proof: Let I, J be as in the proof of the previous theorem.

The decoding proceeds in three steps.

`Step 1` Compute $\tilde{E}_j = E_j + \sum_{i \in I} \tau(c_{ji}) M_i$ for all $j \in J$.

`Step 2` Compute D^{-1}, $D = (\tau(c_{ji})), j \in J, i \in \overline{I}$.

`Step 3` Compute $D^{-1}\tilde{E}$, where $\tilde{E}$ is the matrix whose $(jL+i)^{th}$ row $(i = 0, \ldots, b-1)$ is the i^{th} row of $\tilde{E}_j$.

Remark that the $\tilde{E}_j$'s, E_j's and M_i's are $(L \times w)$-matrices over GF[2] and the $\tau(c_{ij})$'s are $(L \times L)$-matrices over GF[2]. Hence additions and multiplications in the steps above have to interpreted as matrix additions and multiplications.

Assuming $|I| = k$ and $|J| = |\overline{I}| = k$, Step (i) requires $\mathcal{O}(k(m-k)L^2)$ XOR's of words of size w. Step (iii) requires $\mathcal{O}(k^2L^2)$ XOR's. Hence, together these two steps require $\mathcal{O}(mkL^2)$ XOR's of words of size w.

D^{-1} can be computed by first computing the inverse of $(c_{ji}), j \in J, i \in \overline{I}$, over GF[$2^L$], and then replacing each element in the resulting matrix by its image under τ. Since D is a $(k \times k)$-matrix, by Theorem 5.3 this requires $\mathcal{O}(k^2)$ arithmetic operations in GF[2^L]. ◻

In `Step 2`, instead of computing D^{-1} using the isomorphism τ and the procedure for inverting Cauchy-matrices we could also use Gaussian elimination over GF[2]. This would require $\mathcal{O}((m-k)^3L^3)$ bit operations. It turns out that in practice the method described in the proof of Theorem 6.4 is faster.

7 Implementation details and timing information

The XOR-code was implemented in C, and runs on most Unix platforms, including HP workstations running HP/UX and Sun workstations running SunOS and Solaris.

The driver used to test the program was structured as follows:

- Generate a random message.
- Initialize the field structures needed by the erasure-resilient code.
- Encode the data into packets using the XOR-based erasure-resilient code.
- Destroy some of these packets.
- Decode the message from the remaining packets.
- Compare the retrieved message to the original to ensure that the decoding was correct.

The main computation time is spent in the encoding and decoding routines. Less than 1% of the time is spent in the other parts of the program. The field initialization consists of building two tables that allow us to go back and forth between the following two possible representations of an element of the finite field GF[2^L]:

- The exponent to which one needs to raise a given generator of multiplicative group of the finite field to obtain the particular element. This representation is very useful to multiply two elements, as one just needs to add their exponents (modulo $2^L - 1$).
- As an L-dimensional vector over GF[2]. This representation is useful to add elements, as well as to generate the matrix representation of the element.

7.1 Encoding

The encoding routine consists of three steps:

- Set the identifier in each packet being sent, so that, on the decoding side, the packets can be identified.
- Copy the appropriate message parts into the information packets.
- Compute the values of the redundant packets.

More than 99% of the time is spent in computing the contents of the redundant packets, as the first two steps are just linear in the number of packets sent. To compute the redundant packets, we create a Cauchy matrix over GF[2^L], and replace the elements of this matrix by the matrix representation described in Section 4. This gives us the generator matrix of the XOR-code.

The parameters for computing the redundant packets are:

Mpackets This is the number of message packets sent.

Rpackets This is the number of redundant packets sent.

Lfield This will also be referred to as L. The field we use is GF[2^L].

Nsegs This is the number of segments in our packet. Since we use a word size of 32 bits the size of our packets is: $32 \cdot L \cdot$ Nsegs bits.

The redundant packets are computed as follows:

```
/* The variables that are defined prior to computing the redundant
   packets are:
   MultField:  The size of the multiplicative group of the finite field.
   ExpToFieldElt:  A table that goes from the exponent of an element, in
                   terms of a previously chosen generator of the
                   multiplicative group, to its representation as a
                   vector over GF[2].
   FieldEltToExp:  The table that goes from the vector representation of
                   an element to its exponent of the generator.
   Bit:  An array of integers that is used to select individual bits:
         (A & Bit[i]) is equal to 1 if the i-th bit of A is 1, and 0
         otherwise.
*/

  For row = 0 to Rpackets-1     /* The number of rows in our Cauchy matrix
                              is equal to the number of redundant
                              packets.  */

    For col = 0 to Mpackets-1     /*  The number of columns in our Cauchy
                                matrix is equal to the number of
                                information packets.  */
```

```
      /* exponent is the multiplicative exponent of the element of the
      Cauchy matrix we are currently looking at. XOR computes the
      coordinatewiswise exclusive-or of the bit representation of two
      integers. % is the remainder operator. */

      exponent =
       (MultField - FieldEltToExp[row XOR col XOR MultField]) % MultField

      For row_bit = 0 to Lfield-1    /*  Each element of our finite field
                                    is now  represented as a Lfield by
                                    Lfield 0-1 matrix. */
        For col_bit = 0 to Lfield-1

          /*  Check if the current bit of the matrix element is 1. */
          If (ExpToFieldElt[exponent+row_bit] & Bit[col_bit])

              For segment = 0 to Nsegs-1

                /* (a^=y) is short for (a = a XOR b). */
                redundant_packets[row][segment+row_bit*Nsegs] ^=
                message[segment+col_bit*Nsegs+col*Lfield*Nsegs]
```

The running time for this algorithm is directly proportional to Mpackets, Rpackets, Nsegs, and L^2. However, we can significantly reduce the running time by computing and storing before the execution of a loop all the values used in that loop that are invariant during the execution of the loop. The further up in the above loop structure that we can compute a value, the greater the performance gain we achieve by eliminating redundant computations. The above algorithm is therefore actually implemented as:

```
  For row = 0 to Rpackets-1
    packet = redundant_packets[row]

    For col = 0 to Mpackets-1

      exponent =
       (MultField - FieldEltToExp[row XOR col XOR MultField]) % MultField

      For row_bit = 0 to Lfield-1
        local_packet = packet + row_bit*Nsegs

        For col_bit = 0 to Lfield-1

          If (ExpToFiedlElt[exponent+row_bit] & Bit[col_bit])
            local_message = message + col_bit*Nsegs + col*Lfield*Nsegs

            For segment = 0 to Nsegs-1
              local_packet[segment] ^= local_message[segment]
```

For typical values of the parameters, such as Mpackets=100, Rpackets=50, Nsegs=25, and L=10, we get a speedup of about 4 times (700 msec vs 3.00 seconds).

The running time of this algorithm is no longer directly proportional to Mpackets, Rpackets, Nsegs, and the L^2, because the operations that are performed early are computed only once for each execution of the inner loops, so that their cost is amortized over each iteration of the inner loops. In particular, for the inner most loop, we find that the cost of the first iteration is much higher than the the cost of following iterations. These iterations reuse many of the values that are computed for the first one. We therefore expect the running time to be an affine function of the number of segments, rather than linear as in the original algorithm.

The savings we get for the two loops depending on Lfield are much more modest, since the operations moved outside the loops are fairly simple. We only get an improvement of about 10% from this, so the running time is still essentially proportional to L^2. From the data we collected for the actual running time of the algorithm, we come up with the following formula for the running time:

$$T(\text{Mpackets}, \text{Rpackets}, L, \text{Nsegs}) = 4.5 \times 10^{-5}\text{msec} \times \text{Mpackets} \times \text{Rpackets} \times L^2 \times (\text{Nsegs}+6).$$

Using the fact that the message size is given by Mpackets $\times$ L $\times$ Nsegs, that the message to packet ratio is given by Mpackets, and that the redundancy is given by Rpackets/Mpackets, we can rewrite the equation above in terms of the parameters MsgSize, which is the message size, Ratio, which is the message to packet ratio, and Red, which is the redundancy (expressed as a fraction). Also note that L will usually be equal to log(Ratio) + 1. The above equation now becomes:

$$T(\text{MsgSize}, \text{Ratio}, \text{Red}, L, \alpha) = 4.5 \times 10^{-5}\text{msec} \times \text{MsgSize} \times \text{Ratio} \times \text{Red} \times L \times (1 + \alpha)$$

where α is a correction factor due to the fact that T is affine in Nsegs: $\alpha = 6/\text{Nsegs}$.

Figure 1 below gives some timing information for the encoding procedure. The number of segments varies between 1 and 100. Mpackets was chosen to be 100, Rpackets is 50 and Lfield is 10. Thus, the message size varies between 32Kbits and 3.2Mbits, and the redundancy is 50%. The information was collected on a Sun SPARCstation 20, with a SuperSPARC Model 61 SPARCmodule CPU's running at 61 MHz, and 64MB of main memory.

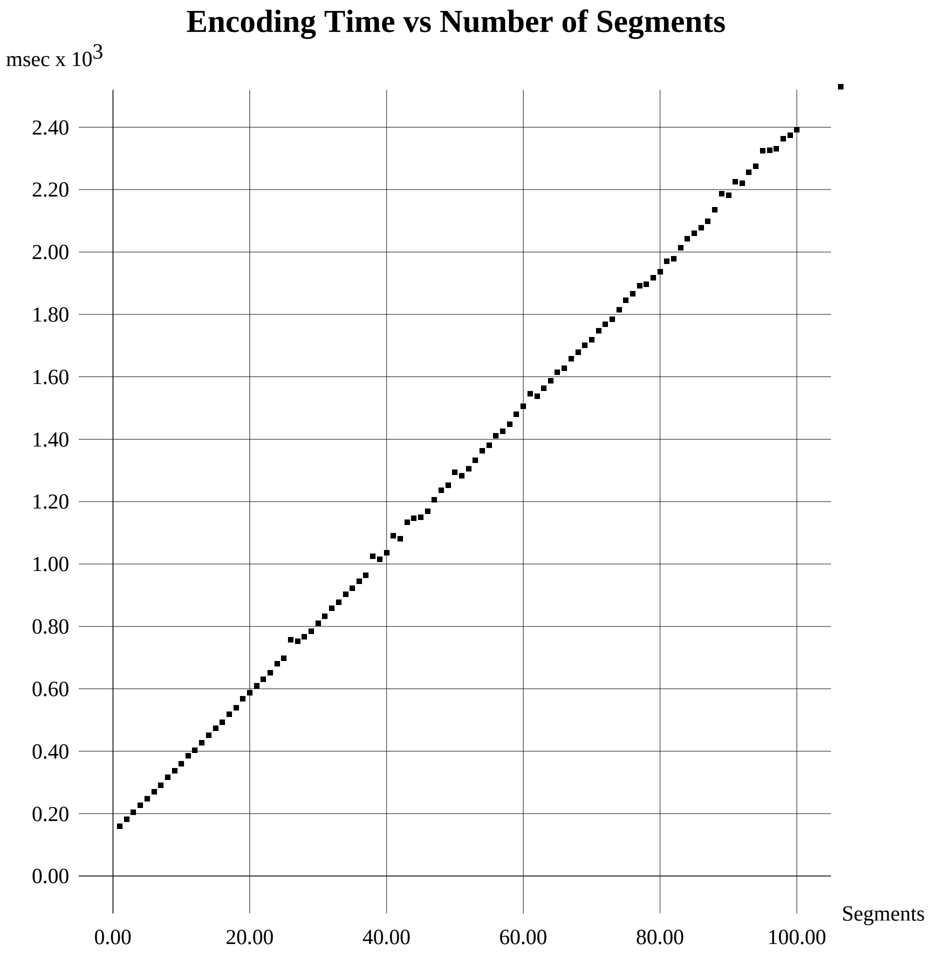

Figure 1

7.2 Decoding

The decoding routine consists of the following steps:

- Check if enough packets were received to recover the message. If not, then return.
- Collect all the information packets that were received, and then pick enough redundant packets to decode the message.
- Copy the information from the information packets into the appropriate parts of the outgoing message.
- Compute the square submatrix M of the original Cauchy matrix over GF[2^L], whose row indices are given by the indices of the redundant packets used and whose column indices are given by the indices of the missing information packets.
- Compute the inverse of M and replace each entry in the matrix by its matrix representation. Call this matrix D.
- Update the contents of each redundant packet as described in Section 6.
- Multiply the updated redundant packets by D to obtain the missing information packets.

For typical input parameters, over 99% of the time is spent in the last three steps. The parameters for the decoding time are:

Mpackets, Lfield, Nsegs These are exactly the same as for the encoding.

Nextra This is the number of extra packets needed to decode the message, which is equal to the number of message packets that were not received. Nextra can range between 0 and Rpackets (the total number of redundant packets sent on the encoding side).

The matrix inversion algorithm is implemented as follows:

```
/*  The variables FieldEltToExp, MultField and MultField are as described
    in the encoding algorithm.

    The following variables are computed in the decoding routine
    before the matrix inversion step is called:
    RowInd:  An array that keeps track of the extra packets received:
              RowInd[i] is the identifier of the i-th extra packet that
              was received.
    ColInd:  An array that keeps track of the message packets received:
              ColInd[i] is the identifier of the i-th message packet
              that was received.
    M:  The submatrix of the original Cauchy matrix.

    The inverted matrix is stored in variable InvMat.  The algorithm we
    use to invert the Cauchy matrix M is explained in Section 4.
*/

  For row = 0 to Nextra-1

    For col = 0 to Nextra-1

      /* != is the Not-Equal operator. */
      If (col != row)

        /* (a+=b) is short for (a = a+b) */
        C[row] += FieldEltToExp[ RowInd[row] XOR RowInd[col] ]
        D[col] += FieldEltToExp[ ColInd[row] XOR ColInd[col] ]

      E[row] += FieldEltToExp[ RowInd[row] XOR ColInd[col] XOR MultField ]
      F[col] += FieldEltToExp[ RowInd[row] XOR ColInd[col] XOR MultField ]


  For row = 0 to Nextra-1

    For col = 0 to Nextra-1

      InvMat[row][col] = E[col] + F[row] - C[col] - D[row]
         - FieldEltToExp[ RowInd[col] XOR ColInd[row] XOR MultField]
      If (InvMat[row][col] >= 0)
        InvMat[row][col] = InvMat[row][col] % MultField

      Else
        InvMat[row][col] =
           (MultField - ( (-InvMat[row][col]) % MultField)) % MultField
```

This algorithm is quadratic in Nextra, and the formula for its running time is:

$$T(\text{Nextra}) = 7.5 \times 10^{-4}\text{msec} \times (\text{Nextra})^2$$

For typical input parameters, this time is negligible compared to the time spent in the steps that update the redundant packets and that eventually retrieve the missing information packets. For instance, if we use values such as Nsegs=25, L=10, Mpackets=100, Rpackets=50, and Nextra=35, the time for the matrix inversion is about 0.2% of the time spent in the overall decoding algorithm (1 msec vs. 500 msec).

The algorithm for the second last step is:

```
/*  Variables MultField, FieldEltToExp, RowInd, MultField, Bit, and
    ExpToFieldElt are as previously defined.

    M is a matrix containing the values from the redundant packets
    that are being used to decode the message.

    RecMsg is the array where the message is to be stored.  The
    information contained in the message packets that were received
    has already been copied to this array.
*/

  For row = 0 to Nextra-1

    For col = 0 to Mpackets-1

      /*  If the message packet was received, then process it. */
      If (RecIndex[col] == 1)
        exponent =
          (MultField - FieldEltToExp[ RowInd[row] XOR col XOR MultField ] )
            % MultField

        For row_bit = 0 to Lfield-1

          For col_bit = 0 to Lfield-1

            If ( ExpToFieldElt[exponent+row_bit] & Bit[col_bit] )

              For segment = 0 to Nsegs-1
```

```
                M[row_bit + row*Lfield][segment] ^=
                  RecMsg[segment + col_bit*Nsegs + col*Lfield*Nsegs]
```

The algorithm for the last step is:

```
/*  InvMat is the inverted matrix computed during the matrix inversion
    step.

    ExpToFieldElt, Bit, RecMsg, ColInd, and M are as described above.
*/

  For row = 0 to Nextra-1

    For col = 0 to Nextra-1
      exponent = InvMat[row][col]

      For row_bit = 0 to Lfield-1

        For col_bit = 0 to Lfield-1

          If (ExpToFieldElt[exponent+row_bit] & Bit[col_bit])

            For segment = 0 to Nsegs-1

              RecMsg[segment + row_bit*Nsegs + ColInd[row]*Lfield*Nsegs]
                ^= M[col_bit + col*Lfield][segment]
```

Again, these two algorithms were modified by introducing local variables that store whenever possible values that can be computed before the execution of a loop. This results in a similar performance gain to the one we obtained for the encoding. The formula for the total running time for the resulting two algorithms combined is almost exactly the same as the corresponding formula for the encoding algorithm, with Nextra replacing Rpackets. Nextra is bounded above by Rpackets, and in general will be less than Rpackets. For instance, if we send 100 message packets and 50 redundant packets, and we lose 50 packets, we expect to use 33 redundant packets for decoding, which means that the total decoding time will be 2/3 of the encoding time.

The formula for the running time of the last two steps is:

$$T(\text{Mpackets}, \text{Nextra}, L, \text{Nsegs}) = 4.5 \times 10^{-5}\text{msec} \times \text{Mpackets} \times \text{Nextra} \times L^2 \times (\text{Nsegs} + 6)$$

As we did for the encoding, we can rewrite the above equation using the fact that the message size is given by Mpackets $\times$ L $\times$ Nsegs. Using the parameter MsgSize for the message size, we get:

$$T(\text{MsgSize}, \text{Nextra}, L, \alpha) = 4.5 \times 10^{-5}\text{msec} \times \text{MsgSize} \times \text{Nextra} \times L \times (1 + \alpha)$$

where α is a correction factor due to the fact that T is affine in Nsegs: $\alpha = 6/\text{Nsegs}$.

Figure 2 below gives some timing information for the decoding procedure. The number redundant packets used in the decoding procedure varies between 1 and 50. Mpackets was chosen to be 100, Nsegs is 25 and Lfield is 10. Thus, the message is 800Kbits and the redundancy used varies between 1% and 50%. Again the information was collected on a Sun SPARCstation 20, with a SuperSPARC Model 61 SPARCmodule CPU's running at 61 MHz, and 64MB of main memory.

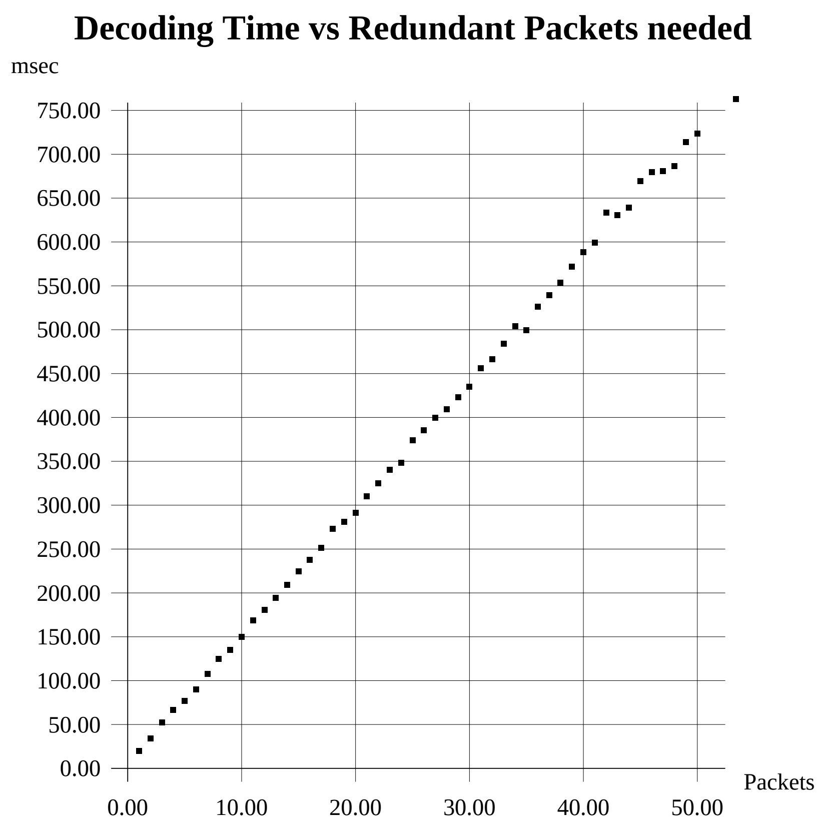

Figure 2

8 A lower bound for the packet size of MDS codes

In this section a lower bound for the packet size of MDS codes is shown. The bound is significantly better than the bound shown in [1]. The bound almost matches the least possible packet size of the Cauchy-code and the XOR-code. For slightly better bounds for the case of special linear MDS codes see [6].

Theorem 8.1 *For any MDS code E that encodes a message consisting of m packets into an encoding $E(M)$ consisting of $n \geq m+2$ packets the packet size b has to satisfy the inequality*

$$2^b + 1 \geq \max\{m, n-m\}.$$

Proof: In the first part it will be shown that if there is an MDS code that encodes messages of m packets of size b into encodings of n packets of size b than there is also a systematic code with the same parameters. Hence it suffices to prove the lower bound for systematic codes. This is done in the second part of the proof.

Without loss of generality we view a packet as describing some element of $S = \{0, \ldots, 2^b - 1\}$.

Let E be an MDS code that maps messages of m packets onto encodings of n packets. The set of messages is S^m and the set of encodings is a subset T of S^n of size 2^{mb}. Since E is an MDS code, any two elements in T agree in at most $m-1$ coordinates. This implies that

(i) Any other bijection of S^m onto T is also an MDS code.

(ii) The projection of the elements in T onto their first m coordinates is a bijection between T and S.

Combining (i) and (ii) shows that the bijection that maps an element M in S^m onto the element in T whose projection onto its first m coordinates is M, is an MDS code. This finishes the first part of the proof.

To prove the lower bound for any systematic code, let E be a systematic code that maps a message $(M_1, \ldots, M_m)$ onto $E(M) = (M_1, \ldots, M_m, E_1(M), \ldots, E_{n-m}(M))$, where each E_i is some function mapping messages onto single packets.

First it is shown that the packet size of this code has to satisfy the inequality

$$2^b + 1 \geq m.$$

For a pair $(x, i), x \in S, x \neq 0, i \in \{1, \ldots, m\}$, let $M(x, i)$ be the message whose i^{th} packet is x and whose remaining packets are 0. For any two distinct pairs $(x, i), (y, j)$

$$(E_1(M(x, i)), E_2(M(x, i))) \neq (E_1(M(y, j)), E_2(M(y, j))). \tag{1}$$

Otherwise the encodings of two different messages $M(x, i)$ and $M(y, j)$ agree in at least m packets and E is not an MDS code.

Since there are $m(2^b - 1)$ pairs $(x, i), x \in S, x \neq 0, i \in \{1, \ldots, m\}$, and each E_j can take on only 2^b different values, (1) implies

$$2^{2b} \geq m(2^b - 1) \;\text{ or }\; 2^b + 1 \geq m. \tag{2}$$

Next it is shown that the packet size of E also has to satisfy

$$2^b + 1 \geq n - m,$$

For any pair $(x, y) \in S^2$ let $M(x, y)$ be the message $(x, y, 0, 0, \ldots, 0)$. Any two of these messages must be distinguishable from the last $m-2$ information packets and any two redundant packets. This implies that for $(x, y) \neq (x', y')$ and $i, j \in \{1, \ldots, n-m\}, i \neq j$

$$(E_i(M(x, y), E_j(M(x, y))) \neq (E_i(M(x', y'), E_j(M(x', y'))). \tag{3}$$

For a fixed $i \in \{1, \ldots, n-m\}$ and for any $k \in S$, let n_k be the number of messages $M(x, y)$ such that $E_i(M(x, y)) = k$. The number of pairs of messages $(M(x, y), M(x', y'))$ such that $E_i(M(x, y)) = E_i(M(x', y'))$ is

$$\sum_{k=0}^{2^b-1} \binom{n_k}{2}.$$

This sum is minimized if $n_k = 2^b$ for all k, in which case its value is $2^b\binom{2^b}{2}$. Hence for each $i \in \{1, \ldots, n-m\}$ there are at least $2^{2b}\binom{2^b}{2}$ pairs of messages $(M(x, y), M(x', y'))$ such that $E_i(M(x, y)) = E_i(M(x', y'))$.

Assume $n - m > 2^b + 1$. This implies

$$\binom{2^{2b}}{2} < (n-m)2^b\binom{2^b}{2}.$$

Since there are only $\binom{2^{2b}}{2}$ different pairs of messages this implies that there is a pair of messages $(M(x,y), M(x',y'))$ such that for two different indices $i, j \in \{1, \ldots, n-m\}$

$$E_i(M(x,y)) = E_i(M(x',y')) \quad \text{and} \quad E_j(M(x,y)) = E_j(M(x',y')).$$

This violates (3). Hence $2^b + 1 \geq n - m$, which finishes the proof. $\square$

For $n = m + 1$ a packet size of 1 suffices. The encoding contains the bits of the message and the XOR of all bits in the message.

References

[1] A. Albanese, J. Blömer, J. Edmonds, M. Luby, M. Sudan, *Priority Encoding Transmission*, in Proc. 35th Symposium on Foundations of Computer Science (FOCS), 1994. pp. 604–613.

[2] A. Albanese, J. Blömer, J. Edmonds, M. Luby, *Priority Encoding Transmission*, Technical Report TR-94-039, International Computer Science Institute, Berkeley, 1994.

[3] N. Alon J. Edmonds, M. Luby, *Linear Time Erasure Codes with Nearly Optimal Recovery*, in Proc. 36th Symposium on Foundations of Computer Science (FOCS), 1995.

[4] E. Biersack, "Performance evaluation of forward error correction in ATM networks", *Proceedings of SIGCOMM '92*, Baltimore, 1992.

[5] D. Grigoriev, M. Karpinski, M. Singer, *Fast Parallel Alogorithms for Multivariate Polynomial over Finite Fields*, SIAM Journal on Computing, Vol. 19, 1990, pp. 1059-1063.

[6] F. J. MacWilliams, N. J. A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, New York, 1977.

[7] L. Mirsky, *An Introduction to Linear Algebra*, Dover, New York, 1982.

[8] M. O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance*, J. ACM, Vol. 36, No. 2, April 1989, pp. 335–348.

[9] S. Roman, Coding and Information Theory, Springer-Verlag, New York, 1992.

[10] A. Shamir, *How to Share a Secret*, C. ACM, Vol. 22, No. 11, November 1979, pp. 612-613.

[11] S. B. Wicker, V. K. Bhargava, *Reed-Solomon Codes and their Applications*, IEEE Press, New York, 1994.