Practical Considerations in Repairing Reed-Solomon Codes

Thi Xinh Dinh†, Luu Y Nhi Nguyen†, Lakshmi J. Mohan†, Serdar Boztas†, Tran Thi Luong‡, and Son Hoang Dau†
†RMIT University, ‡Academy of Cryptographic Technique, Hanoi, Vietnam

I. INTRODUCTION

Reed-Solomon codes [1], invented more than 60 years ago, still constitute the most widely used family of erasure codes in distributed storage systems to date (see Table I). Due to their popularity in practice as well as their fundamental role in the development of classical coding theory, a significant amount of research has been conducted in recent years to improve their performance. In particular, we employ different heuristic algorithms to search for low-bandwidth repair schemes for codes of short lengths with typical redundancies and establish three tables of current best repair schemes for \([n, k]\) Reed-Solomon codes over GF(256) with \(4 \leq n \leq 16\) and \(r = n - k \in \{2, 3, 4\}\). The tables cover most known codes currently used in the distributed storage industry.

II. DEFINITIONS AND NOTATIONS

Let \([n] \triangleq \{1, 2, \ldots, n\}\) and \([m, n] \triangleq \{m, m + 1, \ldots, n\}\). Let \(F_q\) be the finite field of \(q\) elements and \(F_{q'}\) be its extension field of degree \(\ell\), where \(q\) is a prime power. In this work we only consider the case \(q' = 256\). The field \(F_{256}\) can also be viewed as a vector space over its subfields, i.e., \(F_{256} \cong F_{256}^4 \cong F_{256}^4 \cong F_{256}^8\). Each element \(b\) of \(F_{256}\) can be represented as one byte, i.e., a vector of eight bits \((b_1, b_2, \ldots, b_8)\), or an integer in \([0, 255]\).

For instance, \(6 = 2^2 + 2\) is represented by the vector \(00000110\) and corresponds to \(b = 2^2 + z\), where \(z = 2\) is a primitive element of \(F_{256}\). To accelerate the computation over \(F_{256}\)-additions between integers representing finite field elements are performed bitwise while multiplications are based on table lookups. Bitwise operators in C including XOR ‘’, AND ‘&’, and bit-shift ‘’ are heavily used in code optimization.

We use \(\text{span}_{F_q}(U)\) to denote the \(F_q\)-subspace of \(F_{q'}\) spanned by a subset \(U \subseteq F_{q'}\). We use \(\dim_{F_q}(\cdot)\) and \(\text{rank}_{F_q}(\cdot)\) to denote the dimension of a subspace and the rank of a set of vectors over \(F_q\). The (field) trace of an element \(b \in F_{q'}\) over \(F_q\) is \(\text{Tr}_{F_{q'}/F_q}(b) \triangleq \sum_{i=0}^{\ell-1} b_i\). Given an \(F_{q'}\)-subspace \(W\) of \(F_{q'}\), the polynomial \(L_W(x) = \prod_{w \in W}(x - w)\) is called the subspace polynomial corresponding to \(W\).

A linear \([n, k]\) code \(C\) over \(F_{q'}\) is an \(F_{q'}\)-subspace of \(F_{q'}^n\) of dimension \(k\). Each element \(c = (c_1, c_2, \ldots, c_n) \in C\) is referred to as a codeword and each component \(c_j\) is called a codeword symbol. The dual \(C^\perp\) of a code \(C\) is the orthogonal complement of \(C\) in \(F_{q'}^n\) and has dimension \(r = n - k\). The elements of \(C^\perp\) are called dual codewords. We call \(r\) the redundancy of the code. A matrix \(G \in F_{q'}^{n \times r}\) of rank \(k\) over \(F_{q'}\) whose rows are codewords of \(C\) is called a generator matrix of the code.

| Storage Systems          | Reed-Solomon codes | RS(e, k) |
|-------------------------|--------------------|----------|
| IBM Spectrum Scale RAID | RS(10,4), RS(11,8) |          |
| Linux RAID-6            | RS(10,6)           |          |
| Google File System II   | RS(10,6)           |          |
| Quantcast File System   | RS(9,6)            |          |
| Hadoop 3.0 HDF5-EC      | RS(9,6)            |          |
| Yahoo Cloud Object Store| RS(11,8)           |          |
| Facebook’s F4 storage system| RS(14,10)       |          |
| Microsoft’s Pelican cold storage| RS(18,15)      |          |
| Baidu’s Atlas Cloud Storage| RS(12,8)        |          |
| S3 (blockchain-based)   | RS(18,20), RS(80,40), etc. |

TABLE I: A table of Reed-Solomon codes employed in major distributed storage systems - an updated version of [18] Table I.

Despite the growing literature, the treatment of short-length Reed-Solomon codes used in (or relevant to) practical storage systems has been sporadic and rather limited: RS(14,10) (used in Facebook’s f4) has received the most attention [2]–[4], [10], [23], while RS(5,3) and RS(6,4) were investigated in [2], and RS(11,8) and RS(12,8) were discussed in [23]. Apart from [10] and [23], in which they were studied as the main topic of interest, these codes were mostly used as examples to demonstrate the inefficiency of the naive repair and the improvements in repair bandwidths that a carefully designed repair scheme can bring. We address this gap in the literature by providing a systematic investigation of low-bandwidth repair schemes for short-length Reed-Solomon codes that are relevant for the data storage industry and discuss several practical aspects including the selection of the evaluation points and implementation.

We first revisit four constructions of Reed-Solomon codes in existing implementations, observing that all but one are the same as the classical construction using polynomial evaluations (Section III). Next, we discuss heuristic algorithms that can be used for construction of repair schemes for such codes and as a result, establish three tables of current-best repair schemes for codes of length \(n \leq 16\) and redundancy \(r \leq 4\) (Section IV). We also study in this section the impact of the evaluation points on repair bandwidths, demonstrating with an example that codes having the same \(n\) and \(k\) but using different lists of evaluation points may end up having different repair bandwidths. Finally, we propose an efficient way to implement (in C) a trace repair scheme for Reed-Solomon code based on lookup tables and fast bitwise operations [27] on top of the state-of-the-art Intel Intelligent Storage Acceleration Library (ISA-L) (Section V).

Abstract—The issue of repairing Reed-Solomon codes currently employed in industry has been sporadically discussed in the literature. In this work we carry out a systematic study of these codes and investigate important aspects of repairing them under the trace repair framework, including which evaluation points to select and how to implement a trace repair scheme efficiently. In particular, we employ different heuristic algorithms to search for low-bandwidth repair schemes for codes of short lengths with typical redundancies and establish three tables of current best repair schemes for \([n, k]\) Reed-Solomon codes over GF(256) with \(4 \leq n \leq 16\) and \(r = n - k \in \{2, 3, 4\}\). The tables cover most known codes currently used in the distributed storage industry.
Given a generator matrix $G$, a message $\mathbf{u} = (u_1, \ldots, u_k)$ is transformed into a codeword $\mathbf{c} = \mathbf{u}G$. A parity check matrix $H$ of the dual code $C^\perp$.

### III. Existing Constructions of Reed-Solomon Codes

#### A. Classical Construction by Reed and Solomon

The following construction of Reed-Solomon codes is the original one proposed by Reed and Solomon in [1].

**Definition 1.** Let $F_q[x]$ denote the ring of polynomials over $F_q$. A Reed-Solomon code $RS(A, k) \subseteq F_q^n$ of dimension $k$ with evaluation points $A = \{\alpha_1, \ldots, \alpha_k\} \subseteq F_q$ is defined as $RS(A, k) = \{(f(\alpha_1), \ldots, f(\alpha_n)) : f \in F_q[x], \deg(f) < k\}$.

We also use the notation $RS(n, k)$, ignoring the evaluation points. Clearly, a generator matrix of this code is the Vandermonde matrix $\text{Vand}(\alpha_1, \ldots, \alpha_n) \triangleq \begin{pmatrix} \alpha_1^{i-1} & \cdots & \alpha_n^{i-1} \end{pmatrix}_{1 \leq i \leq k, 1 \leq j \leq n}$.

A *generalized* Reed-Solomon code, $GRS(A, k, \lambda \bar{\lambda})$, where $\lambda = (\lambda_1, \ldots, \lambda_n) \in F_q^n$, is defined similarly to a Reed-Solomon code, except that the codeword corresponding to a polynomial $f$ is defined as $(\lambda_1f(\alpha_1), \ldots, \lambda_nf(\alpha_n))$, where $\lambda_j \neq 0$ for all $j \in [n]$. It is well known that the dual of a Reed-Solomon code $RS(A, k)$ is a generalized Reed-Solomon code $GRS(A, n-k, \lambda \bar{\lambda})$, for some multiplier vector $\lambda \bar{\lambda}$ (Chap. 10). We sometimes use the notation $GRS(n, k)$, ignoring $A$ and $\lambda \bar{\lambda}$.

We often use $f(x)$ to denote a polynomial of degree at most $k-1$, which corresponds to a codeword of the Reed-Solomon code $C = RS(A, k)$, and $g(x)$ to denote a polynomial of degree at most $r-1 = n-k-1$, which corresponds to a codeword of the dual code $C^\perp$. Since $\sum_{j=1}^n g(\alpha_j)(\lambda_jf(\alpha_j)) = 0$, we also refer to the polynomial $g(x)$ as a check polynomial for $C$.

Next, we discuss four main constructions of Reed-Solomon codes found in practical systems, three of which are equivalent to the original construction and one is invalid. All are over $F_{256}$.

#### B. Constructions in Intelligent Storage Acceleration Library

Both implementations of Reed-Solomon codes provided by ISA-L are systematic [29]. The first one, also found in the Quantcast File System [30], uses the generator matrix $G = [I_k \mid V]$ in which $V$ is a $k \times (n-k)$ Vandermonde matrix: $V = (x_j^{-1})$, $i \in [1, k], j \in [1, n]$, where $x_j = \zeta_j^{-1}$ and $z = 2$ is a primitive element of $F_{256}$. This is not a construction of a Reed-Solomon code, not even an MDS code (i.e., achieving the Singleton bound [22]). Indeed, it is known that $G = [I_k \mid V]$ generates an MDS code only if every square submatrix of $V$ is invertible ([28], Ch. 11, Thm. 8), which is not true for a Vandermonde matrix in general. Although when $n = 9$ and $k = 6$, as in the Quantcast File System [31], the code is still MDS, we ignore this construction as it is incorrect in general.

We focus on the second construction, which uses the generator matrix $G = [I_k \mid C]$ in which $C$ is a $k \times (n-k)$ Cauchy matrix: $C = \begin{pmatrix} \frac{1}{x_i - y_j} \end{pmatrix}$, $i \in [1, k], j \in [1, n-k]$, where $x_i = i-1$ and $y_j = k+1-j$. Note that $x_i$ and $y_j$ are written using the integer representations of elements of $F_{256}$ and $x_i + y_j$ refers to the (bitwise) addition of two field elements. According to [32] Thm. 1), this code is the same as $GRS(A, k, \lambda \bar{\lambda})$ with $A = [\bar{\lambda} \mid \bar{\mathbf{g}}] = [0, n-1]$ and $\lambda = (\lambda_1, \ldots, \lambda_n)$ defined as $\lambda_j = \begin{cases} 1/\prod_{i=1, i \neq j}^k (x_i - x_j), & 1 \leq j \leq k, \\ 1/\prod_{i=1}^k (x_i + y_j - k), & k + 1 \leq j \leq n. \end{cases}$

For example, for $n = 9, k = 6$, the Cauchy-based construction of RS(9,6) code, or more precisely, a GRS(9,6), uses $\bar{\lambda} = (0, 1, 2, 3, 4, 5)$ and $\bar{\mathbf{g}} = (6, 7, 8)$ and has a generator matrix $G = I_6 \begin{pmatrix} 122 & 186 & 173 \\ 186 & 122 & 157 \\ 71 & 167 & 221 \\ 167 & 71 & 152 \\ 142 & 244 & 61 \\ 244 & 142 & 170 \end{pmatrix}$, $\begin{pmatrix} z_{229} & z_{57} & z_{252} \\ z_{57} & z_{229} & z_{32} \\ z_{253} & z_{205} & z_{204} \\ z_{205} & z_{253} & z_{17} \\ z_{254} & z_{230} & z_{228} \\ z_{230} & z_{254} & z_{151} \end{pmatrix}$, where $z = 2$ is a primitive element of $F_{256}$ satisfying $z^8 + z^4 + z^3 + z^2 + z^0 = 0$ (see the list of Conway polynomials [33]). Note that we are using both the integer representation and the exponential representation of a finite field element. For instance, 122 has the binary representation 01111010, corresponding to $z^6 + z^5 + z^4 + z^3 + z = z^{229}$. This is a GRS(4,6,$\bar{\lambda}$) with $A = [0, 8]$ = {0, 1, $z$, $z^2$, $z^3$, $z^{10}$ = $z^2 + z^0$, $z^{26}$ = $z^2 + z$, $z^{198}$ = $z^2 + z + 1$} and $\lambda = (z^{177}, z^{177}, z^6, z^6, z^{234}, z^{234}, z^{208}, z^{208}, z^{119})$. Using [28] Ch. 10, Thm. 4), we can deduce that the dual of this GRS(4,6,$\bar{\lambda}$) is an GRS(4,3,$\bar{\gamma}$), where $\bar{\gamma} = (z^{47}, z^{82}, z^{171}, z^{239}, z^{221}, z^{144}, z^{75}, z^{199})$. We believe that the selection of $A$ as the first $n$ nonnegative integers in ISA-L is purely for the convenience of the for-loops in its C code and has no significant reasons behind.

#### C. Code Construction in Backblaze Vaults

Backblaze Vaults ([34], [35]) uses a systematic generator matrix $G = [I_k \mid V_1 \mid V_2] = V_1^{-1}V$, where $V = [V_1 \mid V_2]$ is the $k \times n$ Vandermonde matrix created by the finite field elements corresponding to 0, 1, \ldots, $n-1$. In other words, this is a standard RS($A, k$) in which $A = [0, n-1]$, the same set of evaluation points as in the GRS($A, k, \bar{\lambda}$) of ISA-L.

#### D. Code Construction in Facebook's f4

The GRS(14,10) used in Facebook’s f4 storage system [36] has been used repeatedly in the literature to demonstrate either the inefficiency of Reed-Solomon code’s naive repair scheme or improvements from it. At the moment we could no longer locate the official source of the implementation of this code. However, from the previous studies [2]-[4], [10] and from our own copy of the code, GRS(14,10) in is was constructed via the generator polynomial ([22] Ch. 7) $g(x) = (x-1)(x-z)(x-z^2)(x-z^3)$, then the generator polynomial is $g(x) = \prod_{i=0}^{n-1} (x-z^i)$ and the word-weights correspond to vectors of coefficients of all polynomials $c(x) = \sum_{i=0}^{n-1} c_i x^i \in F_{256}[x]$ that admit 1, $z$, $z^2$, \ldots, $z^{n-1}$ as roots, where $r = n - k$. Equivalently, the code has a parity check matrix $H = \text{Vand}(1, z, z^2, \ldots, z^{n-1})$, and hence, it is the dual of an RS($A, k$) with $A = \{1, z, z^2, \ldots, z^{n-1}\}$.

To summarize, all (valid) constructions of (generalized) Reed-Solomon codes in industry available to us, although may look different, are still the same as the original one provided in Section III-A. This is good news because we can focus on just the original construction. Although straightforward, this section serves as a one-stop reference for future studies in this direction.

#### IV. Heuristic Search for Low-Bandwidth Schemes

##### A. Trace Repair Framework

Following the framework developed in [3], [4], a (linear) trace repair scheme for the component $c_{ij}$ of a codeword $\mathbf{c}$
of a linear \([n, k]\) code \(C\) over \(\mathbb{F}_q\) corresponds to a set of \(\ell\) dual codewords \(\{\mathbf{g}^{(i)}\}_{i \in [\ell]} \subset C^\perp\), \(\mathbf{g}^{(i)} = (g^{(i)}_1, \ldots, g^{(i)}_n)\), satisfying the Full-Rank Condition: \(\text{rank}_{q} \{\mathbf{g}^{(i)}: i \in [\ell]\} = \ell\). Such a repair scheme is denoted \(R(\{\mathbf{g}^{(i)}: i \in [\ell]\})\), which can be viewed as an \(\ell \times n\) repair matrix with \(g^{(i)}_j\) as its \((i, j)\)-entry. As established in [3], [4], the repair bandwidth of such a repair scheme (in bits) is \(b(R) = \sum_{j \in [n]} \sum_{\{i \in [\ell] : g^{(i)}_j \neq 0\}} r_j\), where \(r_j \coloneqq \text{rank}_{q} \{\mathbf{g}^{(j)}: i \in [\ell]\}\). To repair all \(n\) components of \(\mathbf{c}\), we need \(n\) such repair schemes (possibly with repetition). See, e.g., [22], for a detailed explanation of why the above scheme works with an example. We describe an implementation of this repair scheme later in Section VII.

Note that as the dual of a (generalized) Reed-Solomon code is another generalized Reed-Solomon code, searching for a set of dual codewords is equivalent to searching for a set of polynomials \(\{g_i(x): i \in [\ell]\} \subset \mathbb{F}_q[x]\) of degree at most \(r - 1\). Using the notation above, we have \(g_i(x) = \lambda_i g_i(\alpha_j)\). As \(\lambda_i\)'s do not affect the repair bandwidth, we usually ignore them.

B. Heuristic Search for Low-Bandwidth Repair Schemes

As discussed in Section III and Section VIII-A, to construct low-bandwidth repair schemes for Reed-Solomon codes over \(\mathbb{F}_{256}\), one must find sets of eight check polynomials \(\{g_i(x): i \in [8]\} \subset \mathbb{F}_{256}[x]\) that satisfy the Full-Rank Condition while incurring low repair bandwidths.

There are two main types of check polynomials applicable for Reed-Solomon codes over \(\mathbb{F}_{256}\): the algebraic ones are based on algebraic structures such as subfields and subspaces [5], [4], [9], [15], [22], [23] or cosets of subfields [11], [12], while the others are found by a computer search [3], [10]. In some special cases, for example when \(r = 4\) and \(n \geq 12\) (e.g., GRS(14,10) or GRS(12,8)), the algebraic constructions generate the lowest known repair bandwidths [23]. But for many other cases, an algebraic construction only yields an insignificant reduction from the naive bandwidth, e.g., 6% when \(r = 3\) and \(n = 11\), whereas a computer search can produce a scheme achieving a much higher reduction, e.g., 28% in this case (see Table III).

Note that there are many different RS(n, k)’s depending on which set of evaluation points \(A\) is chosen (we will show later that different \(A\)’s may lead to different (optimal) repair bandwidths). In this work, we examine two types of codes:

- ISAL-codes: \(A = [0, n - 1] \subset \mathbb{F}_{256}\), \(n \leq 256\).
- \(\mathbb{F}_{16}\)-based codes: \(A = \{0, 1, z_{16}, \ldots, z_{16}^{n-2}\} \subset \mathbb{F}_{16}\), \(n \leq 16\), where \(z_{16}\) is a primitive element of \(\mathbb{F}_{16}\) satisfying \(z_{16}^2 + z_{16} + 1 = 0\) (see the list of Conway polynomials [33]).

Searching for good repair schemes for ISAL-codes is much harder because of the very large search space (over \(\mathbb{F}_{256}\)). For \(\mathbb{F}_{16}\)-based codes, the search complexity is lower, which allows us to locate good schemes within a reasonable amount of time.

The “lifting” technique, which previously has been employed only in the context of algebraic constructions [11]–[14], [23], is now used in our heuristic algorithms to transform a repair scheme for codes over \(\mathbb{F}_{16}\) to a repair scheme for codes over \(\mathbb{F}_{256}\) (Proposition 2). We observe that the current-best repair bandwidths of \(\mathbb{F}_{16}\)-based codes are always at least as low as those of ISAL-codes and in many cases are smaller. Showing that this is true in general for Reed-Solomon codes over \(\mathbb{F}_{256}\) (or finding a counterexample) is an interesting open problem.

To find the lowest-bandwidth repair scheme, in general, we need to examine \(\binom{P}{\ell}\) different sets of \(\ell\) polynomials each, where \(P\) is the number of candidate check polynomials. This number is huge even for very modest parameters. To reduce the search complexity, one needs to reduce \(P\) and/or \(\ell\). We discuss different ways to achieve complexity reduction below.

To reduce \(P\), the number of candidate check polynomials, we make the following simplifying assumptions (see also [10]):

- (A1) \(\deg(g_i) = r - 1\): we prove in Proposition 1 that this assumption does not lead to suboptimal solutions, and
- (A2) \(g_i(x)\) has \(r - 1\) (possibly repeated) roots in \(A\): this is based on the (unproven) intuition that more zeros in the repair matrix \(R\) may lead to a low repair bandwidth. This has been confirmed empirically in our various experiments.

**Proposition 1.** Let \(\{g_i(x)\}_{i \in [\ell]}\) be a set of check polynomials of degree at most \(r - 1\) corresponding to a repair scheme for \(c_{j^*}\) of an RS(A, k) over \(\mathbb{F}_q\). Then there exists another set of check polynomials \(\{h_i(x)\}_{i \in [\ell]}\) of degree exactly \(r - 1\) that can repair \(c_{j^*}\) with the same or smaller bandwidth.

**Proof.** Case 1. If one polynomial has degree exactly \(r - 1\), e.g., \(\deg(g_i) = r - 1\), then we set

\[h_i(x) = \begin{cases} g_i(x), & \text{if } \deg(g_i) = r - 1, \\ g_i(x) + g_i(x), & \text{if } \deg(g_i) < r - 1. \end{cases}\]

Then \(\deg(h_i) = r - 1\) for every \(i\) and moreover,

\[\text{span}_{q} \{h_i(\alpha)\}_{i \in [\ell]} = \text{span}_{q} \{g_i(\alpha)\}_{i \in [\ell]},\]

for every \(\alpha \in A\). Thus, \(\{h_i(x)\}_{i \in [\ell]}\) is another repair scheme for \(c_{j^*}\) and has the same bandwidth as \(\{g_i(x)\}_{i \in [\ell]}\).

Case 2. If \(\deg(g_i) < r - 1\) for every \(i \in [\ell]\) then we select an \(\overline{\alpha} \in A \setminus \{\alpha_{j^*}\}\) and set \(h_i(x) = g_i(x)(x - \overline{\alpha})^{r - 1 - \max\{\deg(g_i)\}}\).

Then \(\{h_i(x)\}_{i \in [\ell]}\) is another repair scheme for \(c_{j^*}\) and has the same or smaller bandwidth. Moreover, at least one \(h_i\) has degree exactly \(r - 1\), which reduces this case to Case 1.

To reduce \(\ell\), the number of polynomials needed in a search, we use the well-known lifting and extension techniques. The lifting technique allows us, for instance, to transform a repair scheme for a Reed-Solomon code constructed in \(\mathbb{F}_{16}\) to a repair scheme for another Reed-Solomon code constructed in \(\mathbb{F}_{256}\) using the same set of evaluation points while doubling the bandwidth. The extension technique allows us, for example, to transform a repair scheme of a Reed-Solomon code over \(\mathbb{F}_{256}\) with the base field \(\mathbb{F}_4\) into a repair scheme with base field \(\mathbb{F}_2\). For completeness, we formalize these techniques below.

**Proposition 2 (Lifting).** Suppose that \(m | \ell\) and \(\{g_i(x)\}_{i \in [m]} \subset \mathbb{F}_q[x]\) corresponds to a repair scheme with bandwidth \(b\) (measured in elements in \(\mathbb{F}_q\)) for the \(j^*\)-th component of an RS(A, k) over \(\mathbb{F}_{q^m}\). Then \(\{\beta_j g_i(x)\}_{i \in [m], j \in [\ell/m]}\), where \(\{\beta_j\}_{j \in [\ell/m]}\) is an \(\mathbb{F}_{q^m}\)-basis of \(\mathbb{F}_{q^m}\), corresponds to a repair scheme with bandwidth \(\frac{\ell}{m}b\) for the \(j^*\)-component of the RS(A, k) with the same evaluations points \(A\) but constructed over \(\mathbb{F}_{q^m}\).

**Proposition 3 (Extension).** Suppose that \(m | \ell\) and the set \(\{g_i(x)\}_{i \in [\ell/m]} \subset \mathbb{F}_q[x]\) corresponds to a repair scheme with bandwidth \(b\) (measured in elements in \(\mathbb{F}_q\)) for the component \(c_{j^*}\) of an RS(A, k) over \(\mathbb{F}_q\), treating \(\mathbb{F}_{q^m}\) as the base field. Then the set \(\{g_j(x)\}_{i \in [\ell/m], j \in [m]}\), where \(\{g_j\}_{j \in [m]}\) is an \(\mathbb{F}_{q^m}\)-basis of \(\mathbb{F}_q^m\), corresponds to a repair scheme with bandwidth \(bm\) (measured in elements in \(\mathbb{F}_q\)) for the \(c_{j^*}\) of the same code.
TABLE II: Current-best repair bandwidths (measured in bits) for Reed-Solomon codes with 4 ≤ n ≤ 16 and r = 2.

Applying the above complexity reduction assumptions and techniques, we construct low-bandwidth repair schemes for ISAL-codes and $F_{16}$-codes utilizing the following algorithms.

- **Algorithm 1**: (degree-four repair) Introduced in [10] to tackle GRS(14,10) over $F_{256}$, this algorithm first constructs a list of pairs of polynomials in $F_{q^r}[x]$ (treating $F_{q^{r/2}}$ as the base field) that has bandwidth at most a threshold $\theta_2$, and then search for sets of four polynomials (treating $F_{q^{r/4}}$ as the base field) consisting of two pairs from that list (keeping the first pair unchanged while adding a multiplicative factor to the second) that has bandwidth at most $\theta_4$ ($\theta_4 < \theta_2$). Various thresholds $\theta_2$ and $\theta_4$ were tested to produce the lowest bandwidth.

- **Algorithm 2**: (exhaustive search) For $r = 2, 3$, we can also apply a direct exhaustive search for sets of polynomials with low-bandwidths. Note that we do not have to wait for the algorithm to finish (which would take too long). We can retrieve the currently found bandwidths for all codeword components and stop if they find them satisfactory or see that there is little chance to improve further.

Note that both algorithms go through each valid set of polynomials *once* and check which codeword components could be repaired by the set. Algorithm 1 terminates if repair schemes of bandwidth not exceeding the specified threshold have been found for all codeword components. Best found bandwidths for codes with 4 ≤ n ≤ 16 and 2 ≤ r ≤ 4 are reported in Table II, Table III, and Table IV. Column “$F_{16}$-based algebraic” refers to repair schemes using subspace polynomials and lifting [12]. $F_{16}$-based heuristic always finds the best bandwidths, which could also be due to the fact that it is cheaper to search over $F_{16}$ than $F_{256}$. We maintain a web page [37] to keep track of the best bandwidths and the corresponding repair schemes.

**Definition 2** (Bandwidth profile). Given $n > 0$ and $\ell > 0$, a bandwidth profile $\bar{b} = (b_1, b_2, \ldots, b_\ell)$, $b_j \in \{0, 1\}$, is feasible for an RS($A, k$), where $A \subseteq F_q^*$, $|A| = n$, if there exists a collection of $n$ repair schemes that require bandwidth $b_j$ for the $j$-th components of its codeword, $j \in [n]$. A bandwidth profile $\bar{b}^*$ is optimal if for every $j \in [\ell]$, $b_j$ is the lowest bandwidth possible to (linearly) repair the $j$-th codeword component of the code.

Note that in Table II, Table III, and Table IV we report $\max(\bar{b}) \triangleq \max\{b_j : j \in \ell\}$, where $\bar{b}$ is a feasible bandwidth profile. Individual components may require lower bandwidths. Visit our web page [37] for the most updated bandwidths.

| Redundancy | Default | ISA-L heuristic | $F_{10}$-based algebraic | $F_{10}$-based heuristic |
|------------|---------|-----------------|--------------------------|--------------------------|
| r = 2      | 16      | 12 (-25%)       | 18 (+12.5%)              | 12 (-25%)                |
| n = 4      | 24      | 18 (-25%)       | 24 (+0%)                 | 18 (-25%)                |
| n = 5      | 32      | 24 (-25%)       | 30 (-6.3%)               | 24 (-25%)                |
| n = 6      | 40      | 32 (-20%)       | 36 (-10%)                | 30 (-25%)                |
| n = 7      | 48      | 38 (-20.8%)     | 42 (-12.5%)              | 36 (-12.5%)              |
| n = 8      | 56      | 44 (-21.4%)     | 48 (-14.3%)              | 44 (-21.4%)              |
| n = 9      | 64      | 50 (-21.9%)     | 54 (-15.8%)              | 50 (-21.9%)              |
| n = 10     | 72      | 58 (-19.4%)     | 60 (-16.7%)              | 56 (-22.2%)              |
| n = 11     | 80      | 64 (-20%)       | 66 (-17.5%)              | 62 (-20%)                |
| n = 12     | 88      | 72 (-18.2%)     | 72 (-18.2%)              | 70 (-20.5%)              |
| n = 13     | 96      | 80 (-16.7%)     | 88 (-18.8%)              | 76 (-20.8%)              |
| n = 14     | 104     | 88 (-20.2%)     | 84 (-19.2%)              | 84 (-19.2%)              |
| n = 15     | 112     | 90 (-19.6%)     | 90 (-19.6%)              | 90 (-19.6%)              |

TABLE III: Current-best repair bandwidths (measured in bits) for Reed-Solomon codes with 4 ≤ n ≤ 16 and r = 3.

**Proposition 4.** Translating and dilating the set of evaluation points of a Reed-Solomon code do not affect its optimal bandwidth profile. In other words, RS($A, k$) and $RS(\beta A + \gamma, k)$, where $\beta, \gamma \in F_q, \beta \neq 0$, have the same optimal bandwidth profile (up to a permutation).

**Proof.** Suppose that $\{g_i(x)\}_{i \in [\ell]}$ is a repair scheme for RS($A, k$), which can be used to repair $f(\alpha x)$, $\alpha \in A$, and has bandwidth $b$. We set $h_i(x) \triangleq g_i((x - \gamma)/\beta)$, $i \in [\ell]$. Then $h_i(\beta \alpha + \gamma) = g_i(\alpha)$ for every $\alpha \in A$. Therefore, $\{h_i(x)\}_{i \in [\ell]}$ can be used to repair $\beta \alpha x + \gamma \in \beta A + \gamma$ with bandwidth $b$. Hence, $\beta \alpha + \gamma \in \beta A + \gamma$ can be repaired in $RS(\beta A + \gamma, k)$ with a bandwidth not exceeding that for $\alpha \in A$ in $RS(A, k)$. Since $A = \beta^{-1}(\beta A + \gamma) - \beta^{-1}\gamma$, the same argument proves that the reverse conclusion is also true. Thus, these two codes have the same optimal repair bandwidth for their corresponding codeword components ($f(\alpha x)$ and $f(\beta \alpha x + \gamma)$).

As a corollary of Proposition 4, although there are $q^n$ different RS($n, k$) codes over $F_q^*$, we can divide them into classes of codes that have evaluation points obtained from each other by translations and dilations. Each class can have at most $q^\ell (q^\ell - 1)$ members with the same optimal bandwidth profile (up to permutations). Identifying other transformations of $A$ that preserve the optimal bandwidth profile is an open problem.

Table IV: Current-best repair bandwidths (measured in bits) for Reed-Solomon codes with 5 ≤ n ≤ 16 and r = 4.

| Redundancy | Default | ISA-L heuristic | $F_{10}$-based algebraic | $F_{10}$-based heuristic |
|------------|---------|-----------------|--------------------------|--------------------------|
| r = 4      | 8       | 8 (-0%)         | 10 (±100%)               | 8 (-0%)                  |
| n = 5      | 16      | 12 (-25%)       | 20 (+25%)                | 12 (-25%)                |
| n = 6      | 24      | 16 (-33.3%)     | 24 (+0%)                 | 16 (-33.3%)              |
| n = 7      | 32      | 22 (-31.3%)     | 28 (-12.5%)              | 22 (-31.3%)              |
| n = 8      | 40      | 28 (-30%)       | 32 (-20%)                | 26 (-35%)                |
| n = 9      | 48      | 36 (-25%)       | 36 (-25%)                | 32 (-33.3%)              |
| n = 10     | 56      | 42 (-25%)       | 40 (-28.6%)              | 38 (-32.1%)              |
| n = 11     | 64      | 48 (-22.5%)     | 44 (-31.3%)              | 44 (-31.3%)              |
| n = 12     | 72      | 54 (-25%)       | 48 (-33.3%)              | 48 (-33.3%)              |
| n = 13     | 80      | 62 (-22.5%)     | 52 (-35%)                | 52 (-35%)                |
| n = 14     | 88      | 68 (-22.7%)     | 56 (-36.4%)              | 56 (-36.4%)              |
| n = 15     | 96      | 70 (-37.5%)     | 60 (-37.5%)              | 60 (-37.5%)              |

C. The Impact of Evaluation Points

In this section we discuss important issues regarding the selection of the evaluation points in Reed-Solomon codes. In particular, we demonstrate via an example that different sets of evaluation points may lead to different repair bandwidths.
To examine the impact of evaluation points on repair bandwidths, we consider RS(5,3) codes over $\mathbb{F}_{16}$, which have small $r$ and field size and hence allow us to determine their optimal bandwidth profiles. An RS(5,3) (with a given generator matrix) was first investigated in [2]. We searched through all 524,160 different arrangements of five elements in $\mathbb{F}_{16}$ and found 240 different $A$’s (orders of elements are important) giving rise to the same code, e.g., $A = \{0, 1, z_{16}, z_{16}^2, z_{16}^3\}$, where $z_{16}$ is a primitive element of $\mathbb{F}_{16}$ satisfying $z_{16} + z_{16} + 1 = 0$. It was shown in [2] that the optimal repair bandwidth for each systematic component ($j = 1, 2, 3$) is 10 bits. On the other hand, from Table II and [37], we found that 9 bits are sufficient to repair other RS(5,3) codes. This shows that different sets of evaluation points can lead to different repair bandwidths.

In fact, we have obtained a complete picture of the bandwidth profiles of all RS$(n,n-2)$ codes, 4 $\leq n \leq 16$, over $\mathbb{F}_{16}$. First, from the list of 16 monic polynomials of degree one over $\mathbb{F}_{16}$, we created a list of 6,142,500 = $\binom{16}{4}15^3$ sets of four check polynomials (keeping the first monic while adding arbitrary nonzero coefficients to others). We then generate a rank profile $r = (r_1,r_2,\ldots,r_{16})$ for each polynomial set and remove those whose components are all smaller than 4, which took our GAP program 10 hours to complete. More than six millions sets of polynomials remain as potential repair schemes. For each $A \subseteq \mathbb{F}_{16}$, we determine the optimal repair bandwidth for each codeword component of RS$(A,k)$ by going through all the rank profiles, restricting to the positions in $A$. For instance, when $n = 5$ and $k = 3$, among 4368 = $\binom{15}{3}$ different 5-subsets $A \subseteq \mathbb{F}_{16}$, 2880 have bandwidth profile (9,9,9,9,8), 1440 have bandwidth profile (9,9,9,9,9), while only 48 have bandwidth profile (10,10,10,10,10). It turns out that the RS(5,3) examined in [2] is accidentally among the minority that require 10 bits. Most others need only 9 bits.

V. AN IMPLEMENTATION OF TRACE REPAIR SCHEMES IN C

We implemented a trace repair scheme [27] on top of the existing C code in the ISA-L [29]. As the bandwidth reduction is known, we focus on minimizing the computational complexity. Following our notation in the C code, Node $i$ aims to recover $c_j$ by downloading relevant data from Nodes $i, i \in N \setminus \{j\}$. In ISA-L $|I| = k$ while in our implementation $|I| = n - 1$. The code implemented in ISA-L is systematic, i.e., the first $k$ components are data and the last $n - k$ components of each codeword are parities. Finite field elements over $\mathbb{F}_{256}$ are represented as integers in [0, 255] (see Section II).

A. ISA-L IMPLEMENTATION

ISA-L uses the naive scheme to repair the generalized Reed-Solomon codes, which means that each helper Node $i$ simply reads and sends $c_i$ without performing any computation. Node $j$, after receiving data from $k$ nodes, can recover $c_j$ as follows. A lost parity component can be repaired by downloading the data components (systematic part) and performing encoding. A missing data component, however, requires more work: Node $j$ performs first a matrix inversion to obtain $(G[I])^{-1}$, where $G[I]$ denotes the submatrix of $G$ consisting of columns of $G$ indexed by $I$, and then multiply this matrix with the vector consisting of $c_i, i \in I$. This procedure is repeated for $T$ different codewords, where $T$ is a large number representing the number of codewords to be repaired. For instance, if the encoded file

| Table V: The running times of the naive repair (ISA-L) and trace repair for an RS(9,6) over ten millions codewords and (random) single erasures on a Linux server: Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz with 792 GB RAM. For senders, the maximum running time among all helpers was used. |
|-----------------------------------------------|-----------------|-----------------|-----------------|
|                               | Senders | Receiver | Total     |
| ISA-L (naive)                  | 0.5 (sec)| 0.57 (sec) | 0.57 (sec) |
| Trace repair                   | 0.2 (sec)| 1.4 (sec)  | 1.6 (sec)  |

B. OUR IMPLEMENTATION

In a trace repair scheme, as opposed to ISA-L, both senders and receiver perform computations. As a large number of codewords are being repaired, say, millions, it is crucial to identify parts of the computation that could be precomputed and stored for fast access. We convert the $n$ repair schemes (for $n$ components) into three lookup tables: $H$ (helper) allows the helper nodes to create the repair traces (bits) while $R$ (repair) and $D$ (dual basis) allow the receiver node to process the repair traces and recover the lost component. Please refer to [22] for undefined terminologies. A frequently used operation is the XOR-sum of the bits of an integer. Hence, we precompute an array called Parity that store these values for all $m \in [0, 255]$.

We describe below the computation for one codeword.

**Sender side.** Node $i$ extracts the number of traces to be sent $r_i = H[i][s][0]$, and uses $r_i$ numbers $H[i][s][1] \to H[i][s][r_i]$ to compute $r_i$ repair traces. Table $H$ is defined in a way that the $s$-th trace from Node $i$ is the inner product of $H[i][s][s] \& c_i$.

**RepairTrace**$_i[s] = \text{Parity}[H[i][s][s] \& c_i], \ s \in [1, r_i]$.

Node $i$ then sends $\text{RepairTrace}_i$ to Node $j$.

**Receiver side.** Node $j$ uses $\text{RepairTrace}_i, \ i \in N \setminus \{j\}, R,$ and $D$ to recover $c_j$. As follows. For each $i$, it generates eight column traces $\text{ColumnTrace}_i[s], \ s \in [8]$, using the formula

$\text{ColumnTrace}_i[s] = \text{Parity}[R[i][s][s] \& \text{Dec}(\text{RepairTrace}_i)],$

where $\text{Dec}(\text{RepairTrace})$ turns the $r_i$ bits in $\text{RepairTrace}_i$ into a decimal number, ready for bitwise operations. $\text{Dec}()$ is also implemented using bit-shift and XOR operations. Finally,

$c_j = \oplus_{s=1}^{8}(\oplus_{i \in N \setminus \{j\}} \text{ColumnTrace}_i[s] \times D[s]).$

In our implementation [27], we further optimize the code by joining the above steps at the receiver to save time. For RS(9,6), our implementation is 2.8x slower than ISA-L (Table V). For other codes such as RS(11,8), RS(16,13), RS(12,8), RS(14,10), RS(16,12), the gap is 1.8x-2.4x. Despite being a negative result, the small gaps are encouraging because trace repair is inherently more complicated. Further optimizations may also reduce the gaps. As shown in [38] Fig. 1, the network transfer time is much larger than the computation time (say, 40 times) in naive repair. Hence, the reduction in bandwidth can well compensate the computation time and make trace repair faster. For example, a combination of 33% reduction in bandwidth and 200% increase in computation time could still lead to a 1.25x speed-up compared to naive repair. Implementing trace repairs on Hadoop 3 to verify this observation is a future work.

ACKNOWLEDGEMENT

This work has been supported by the 210124 ARC DECRRA Grant DE180100768. We thank Nguyen Dinh Quang Minh and Dau Trung Dung for their help in the implementation.
