

Short Paper: LESS is More for I/O-Efficient Repairs in Erasure-Coded Storage

Paper ID: 154

Abstract

I/O efficiency is critical for erasure-coded repair performance in modern distributed storage. We propose LESS, a family of repair-friendly erasure code constructions that reduces both the amount of data accessed and the number of I/O seeks in single-block repairs, while ensuring balanced reductions across blocks. LESS layers multiple extended sub-stripes formed by widely deployed Reed-Solomon coding, and is configurable to balance the trade-off between the amount of data accessed and I/O seeks. Evaluation shows that LESS on HDFS reduces both single-block repair and full-node recovery times compared to state-of-the-art I/O-optimal erasure codes.

1 Introduction

Erasure coding is widely implemented in modern distributed storage systems (e.g., [3, 5, 10, 21]) to ensure fault tolerance with significantly lower storage overhead than traditional replication. However, erasure coding incurs high repair costs. Reconstructing a single failed block needs to access and transfer much more data than the block’s size, leading to amplified bandwidth and I/O costs. Extensive research, from both coding theory and systems communities, has proposed repair-friendly erasure code constructions to mitigate bandwidth and I/O costs (see §2.2 for related studies).

I/O efficiency becomes a more critical design factor than bandwidth efficiency in modern distributed storage systems due to rapid advancements in network technologies (e.g., InfiniBand, RDMA, and CXL) for supporting high-speed interconnects, while I/O performance, even with SSDs, cannot match such improvements in improved network speeds especially under random-access workloads. This motivates us to construct repair-friendly erasure codes that make I/O a “first-class citizen” for high repair performance.

However, existing repair-friendly erasure codes often have sub-par I/O performance in repairs (§2.2). For example, Clay codes [36] are state-of-the-art minimum-storage regenerating (MSR) codes that provably minimize the amount of data accessed from local storage in single-block repairs, while minimizing storage redundancy for fault tolerance. Their core idea is *sub-packetization*, which divides an erasure-coded *stripe* (i.e., a set of blocks encoded together) into smaller *sub-stripes* composed of sub-blocks at the same block offset, such that a single-block repair only retrieves a minimum number of sub-blocks across sub-stripes for reconstruction. Despite minimizing data access, Clay codes require exponential sub-packetization for optimality and issue substantial non-contiguous I/O seeks to retrieve sub-blocks in a repair.

This significantly degrades actual repair performance in I/O-bottlenecked environments. Locally repairable codes (e.g., Azure’s LRC [10]) achieve I/O-efficient repairs, but incur higher storage redundancy.

We present LESS, a family of erasure code constructions designed for I/O-efficient repairs, aiming to (i) reduce both the amount of data accessed and the number of I/O seeks with small sub-packetization (as low as *two* sub-stripes per stripe) and (ii) maintain balanced reductions across all blocks within a stripe. LESS builds on the notion of layering extended sub-stripes, by strategically stacking multiple extended sub-stripes (each with a longer stripe length) atop a regular stripe, such that a single-block repair (and some cases of multi-block repairs) can be done within an extended sub-stripe. LESS is designed with practicality in mind. It builds on the widely used Reed-Solomon (RS) codes [30] (§2.1) for a readily understandable design, and preserves the practical properties of RS codes. It further allows configurable sub-packetization to balance the trade-off between data access and I/O seeks.

We implement and evaluate LESS on HDFS [34] in a local cluster. LESS reduces the single-block repair and full-node recovery times of Clay codes by up to 83.3% and 36.6%, respectively. We open-source LESS (anonymously) at <https://anonymous.4open.science/r/less-1E74/>.

2 Background

2.1 Basics of Erasure Coding

We consider a distributed storage system that organizes data in large fixed-size *blocks* (e.g., 128 MiB in HDFS [34] and 256 MiB in Facebook’s f4 [21]) to mitigate I/O overhead. Erasure coding provides fault tolerance against failed blocks. We focus on Reed-Solomon (RS) codes [30], which are widely deployed in production [3, 5, 21, 23]. RS codes are configured by two parameters n and k (where $n > k$). An (n, k) RS code encodes k uncoded data blocks into $n - k$ coded parity blocks, forming a *stripe* of n blocks, and ensures that any k out of n blocks can reconstruct all k data blocks (i.e., any $n - k$ failed blocks are tolerable). In a distributed storage system, multiple stripes are independently encoded, with each stripe distributed across n nodes for tolerating any $n - k$ node failures.

RS codes satisfy three practical properties: (i) *maximum distance separable (MDS)*, i.e., the redundancy ($\frac{n}{k}$ times the data size) is minimized for tolerating any $n - k$ failed blocks; (ii) *general*, i.e., any n and k (where $n > k$) can construct RS codes for a sufficient field size; and (iii) *systematic*, i.e., each stripe keeps k data blocks for direct access.

RS encoding and decoding use linear combinations over a

Galois Field $\text{GF}(2^w)$ with w -bit words, where $n \leq 2^w + 1$ [25]. Each block can be expressed as a linear combination of any k blocks under Galois Field arithmetic (see §3.1 for details).

However, RS codes incur high *repair bandwidth* (i.e., the amount of traffic transferred among nodes during a repair). Repairing a single failed block needs to retrieve k blocks of the same stripe from other nodes; we call this *conventional repair*, which applies to all (n, k) MDS codes. Conventional repair also incurs high *repair I/O* (i.e., the amount of data accessed from local storage during a repair). In high-speed networks, repair I/O becomes the primary bottleneck.

2.2 Related Work on Repair Optimization

Extensive work improves single-block repair performance in erasure-coded storage [1, 33]. We review several representative repair-friendly erasure codes and their limitations.

Minimum-storage regenerating (MSR) codes. MSR codes [4] minimize repair bandwidth for single-block repairs, while preserving the MDS property (i.e., minimum redundancy). They build on *sub-packetization*, which divides each block into $\alpha > 1$ sub-blocks and forms α *sub-stripes* of n sub-blocks at the same offset across blocks. Each sub-block is a linear combination of other $n\alpha - 1$ sub-blocks in the stripe over $\text{GF}(2^w)$. A single-block repair transfers a fraction of sub-blocks, with the minimum repair bandwidth.

Traditional MSR codes [4] require nodes to read and encode all local sub-blocks, leading to high repair I/O. *I/O-optimal MSR codes* minimize repair I/O, by allowing accessed data to be sent directly without encoding. For example, Functional MSR (F-MSR) codes [6, 8] require $n - k = 2$ [8] or $n - k = 3$ [6], with a linear $\alpha = n - k$, but are non-systematic. PM-RBT codes [27] are systematic but require $n \geq 2k - 1$, while Butterfly codes [22] are also systematic but require $n - k = 2$ and incur an exponential $\alpha = 2^{k-1}$.

Clay codes [36] are state-of-the-art I/O-optimal MSR codes that are systematic, support general (n, k) , and are deployed in Ceph [37]. However, Clay codes impose an exponential $\alpha = (n - k)^{\lceil n/(n-k) \rceil}$ (note that exponential sub-packetization is necessary for I/O-optimal MSR codes [2]). Even though Clay codes minimize repair I/O, their repairs access substantial non-contiguous sub-blocks with I/O seeks, leading to sub-par repair performance [35].

Locally repairable codes (LRCs). LRCs [10, 12, 14, 31] reduce repair I/O for single-block repairs. For example, Azure-LRC [10] partitions data blocks into local groups and adds a local parity block per local group. Repairing a failed data or local parity block accesses only the blocks of its local group. However, Azure-LRC still encodes all data blocks of a stripe into global parity blocks, whose repairs still rely on conventional repair. Some LRC variants support local repairs for global parity blocks [14], but LRCs are non-MDS and have higher redundancy than RS codes.

MDS codes with small sub-packetization. To limit I/O

seeks, some MDS codes allow a small α with reduced repair I/O. Hitchhiker codes [29] use $\alpha = 2$ sub-stripes of RS codes by combining sub-blocks across sub-stripes via piggybacking functions, and reduce repair I/O for data blocks by 25-45% compared with RS codes; however, parity-block repairs still follow conventional repair. HashTag codes [16] reduce repair I/O for data blocks with general (n, k) and $\alpha \geq 2$; their extended codes [15] reduce repair I/O for both data and parity blocks, but require $\alpha \geq 4$ and α as a multiple of $n - k$.

Elastic Transformation (ET) [35] can convert an RS code into a repair-friendly code with a configurable $\alpha \geq 2$. However, the transformation limits flexibility from code construction with higher repair improvements.

Repair-efficient algorithms. Our study focuses on erasure code construction, while some studies design repair-efficient algorithms, such as search of minimum-I/O recovery for XOR-based erasure codes [13, 38] or repair parallelization [17, 18, 20, 32]. They do not minimize repair I/O due to constraints from underlying erasure codes.

2.3 Goals

Table 1 summarizes the repair-friendly codes and their design trade-offs, including: non-MDS (e.g., Azure-LRC [10]), restrictive parameters (e.g., F-MSR [6, 8], PM-RBT [27], Butterfly [22]), non-systematic forms (e.g., F-MSR), exponential sub-packetization (e.g., Butterfly, Clay [36]), and data-block-only improvements (e.g., Hitchhiker [29], HashTag [16]).

The limitations inspire LESS, a family of repair-friendly erasure codes with I/O efficiency in mind. We observe that repair performance depends on both repair I/O and I/O seeks. Minimizing repair I/O introduces an exponential α [2, 36] (and hence high I/O seeks) and can negate repair performance gains. LESS targets near-minimum repair I/O with a small and configurable α to limit I/O seeks. Its design goals include:

- **Preserving RS code properties:** LESS is MDS, supports general (n, k) , and remains systematic, as in RS codes.
- **I/O-efficient repair:** LESS achieves I/O-efficient repairs in three aspects: (i) *reduced repair I/O*, (ii) *reduced I/O seeks* (e.g., $\alpha = 2, 3$, or 4); and (iii) *balanced reductions* (i.e., similar reductions in repair I/O and I/O seeks across data and parity blocks). LESS also allows α to be configurable.
- **Single- and multi-block repair efficiency:** Earlier studies focus on optimizing single-block repairs, which dominate in practice (e.g., 98% of failures in $(14, 10)$ -coded stripes [28]). However, wide-stripe codes (i.e., n and k are large) emerge and make multi-block repairs more common [12]. LESS addresses both scenarios.

3 LESS Design

LESS is constructed with configurable parameters (n, k, α) , where $k < n$ and $2 \leq \alpha \leq n - k$. Its key idea is to layer multiple *extended sub-stripes*, each constructed from sub-blocks and encoded using Vandermonde-based RS codes, such that

Codes	MDS	Parameters	Systematic	Sub-packetization	Reduced repair I/O for data/parity
F-MSR [6, 8]	Yes	$n - k \leq 3$	No	$\alpha = n - k$	Yes
PM-RBT [27]	Yes	$n \geq 2k - 1$	Yes	$k - 1 \leq \alpha \leq n - k$	Yes
Butterfly [22]	Yes	$n - k = 2$	Yes	$\alpha = 2^{k-1}$	Yes
Clay [36]	Yes	general (n, k)	Yes	$\alpha = (n - k) \lfloor \frac{n}{n-k} \rfloor$	Yes
Azure-LRCs [10]	No	general (n, k)	Yes	$\alpha = 1$	No
Hitchhiker [29]	Yes	general (n, k)	Yes	$\alpha = 2$	No
HashTag [16]	Yes	general (n, k)	Yes	$2 \leq \alpha \leq (n - k) \lfloor \frac{k}{n-k} \rfloor$	No
HashTag+ [15]	Yes	general (n, k)	Yes	$4 \leq \alpha \leq (n - k) \lfloor \frac{n}{n-k} \rfloor$	Yes
ET [35]	Yes	general (n, k)	Yes	$2 \leq \alpha \leq (n - k) \lfloor \frac{n}{n-k} \rfloor$	Yes
LESS	Yes	general (n, k)	Yes	$2 \leq \alpha \leq n - k$	Yes

Table 1: Comparisons of repair-efficient erasure codes.

a single-block repair always retrieves the sub-blocks from a single extended sub-stripe. LESS also enables I/O-efficient multi-block repairs if the failed blocks share the same extended sub-stripe.

3.1 Vandermonde-Based RS Codes

LESS builds on Vandermonde-based RS codes. An (n, k) Vandermonde-based RS code uses a *parity-check matrix* [24] to define the relationships between k data blocks B_1, B_2, \dots, B_k and $n - k$ parity blocks B_{k+1}, \dots, B_n in $\text{GF}(2^w)$ based on the *parity-check equation*:

$$[\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n] \cdot [B_1, B_2, \dots, B_n]^T = \sum_{i=1}^n B_i \mathbf{v}_i = \mathbf{0}, \quad (1)$$

where $\mathbf{v}_i = [1, v_i, v_i^2, \dots, v_i^{n-k-1}]^T$ is a column vector of length $n - k$, and v_i is a distinct coefficient in $\text{GF}(2^w)$ associated with block B_i , for $1 \leq i \leq n$. In $\text{GF}(2^w)$, an addition is equivalent to a bitwise-XOR operation. The $(n - k) \times n$ parity-check matrix $[\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n]$ ensures that any $n - k$ out of n columns are linearly independent, so as to allow reconstruction of any $n - k$ blocks from the remaining k blocks.

The encoding computes the $n - k$ parity blocks from the k data blocks based on Equation (1):

$$[B_{k+1}, \dots, B_n]^T = [\mathbf{v}_{k+1}, \dots, \mathbf{v}_n]^{-1} [\mathbf{v}_1, \dots, \mathbf{v}_k] [B_1, \dots, B_k]^T. \quad (2)$$

The decoding of any $n - k$ blocks multiplies: (i) the inverted sub-matrix of their associated column vectors, (ii) the sub-matrix of the column vectors associated with the k remaining blocks, and (iii) the k remaining blocks, from Equation (1).

3.2 Motivating Example

We consider LESS via an example. Figure 1 depicts an $(n, k, \alpha) = (6, 4, 2)$ LESS stripe, where each block B_i ($1 \leq i \leq 6$) has two sub-blocks $b_{i,1}$ and $b_{i,2}$. LESS organizes the 12 sub-blocks into three $(8, 6)$ RS-coded extended sub-stripes, X_1, X_2 , and X_3 . Each extended sub-stripe can tolerate any two sub-block failures, and the entire stripe can tolerate any two block failures due to the MDS property.

Construction. We partition the sub-blocks, each of which belongs to exactly two extended sub-stripes. This overlapping

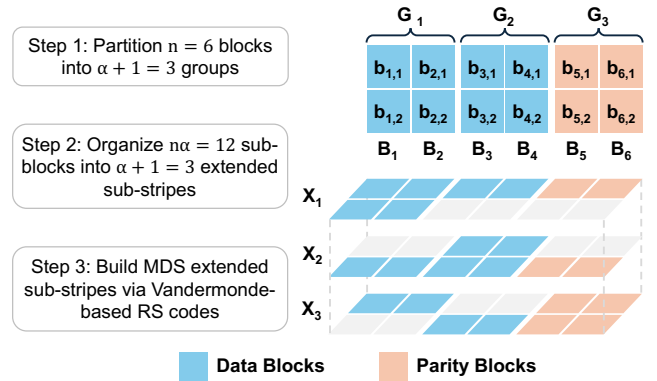


Figure 1: Code design of an $(n, k, \alpha) = (6, 4, 2)$ LESS stripe.

derives each extended sub-stripe from the other two extended sub-stripes. For example, X_3 contains sub-blocks from either X_1 or X_2 , but not both, and can be derived from X_1 and X_2 .

Next, we choose coding coefficients for the extended sub-stripes based on Vandermonde-based RS codes. We select 12 distinct coding coefficients $v_{i,j}$'s for $b_{i,j}$'s from $\text{GF}(2^8)$ ($1 \leq i \leq 6$ and $1 \leq j \leq 2$). We construct X_1 and X_2 as $(8, 6)$ Vandermonde-based RS stripes as:

$$\sum_{i=1}^6 b_{i,1} \mathbf{v}_{i,1} + b_{1,2} \mathbf{v}_{1,2} + b_{2,2} \mathbf{v}_{2,2} = \mathbf{0},$$

$$\sum_{i=1}^6 b_{i,2} \mathbf{v}_{i,2} + b_{3,1} \mathbf{v}_{3,1} + b_{4,1} \mathbf{v}_{4,1} = \mathbf{0},$$

where $\mathbf{v}_{i,j} = [1, v_{i,j}]^T$. X_3 also forms an $(8, 6)$ Vandermonde-based RS stripe, and its parity-check equation for X_3 is the sum of the parity-check equations of X_1 and X_2 :

$$b_{1,1} \mathbf{v}_{1,1} + b_{2,1} \mathbf{v}_{2,1} + b_{3,2} \mathbf{v}_{3,2} + b_{4,2} \mathbf{v}_{4,2} + b_{5,1} \mathbf{v}_{5,1} + b_{5,2} \mathbf{v}_{5,2} + b_{6,1} \mathbf{v}_{6,1} + b_{6,2} \mathbf{v}_{6,2} = \mathbf{0}.$$

Single-block repair. Suppose B_1 fails. LESS repairs $b_{1,1}$ and $b_{1,2}$ from X_1 by retrieving six sub-blocks $b_{2,1}, b_{2,2}, b_{3,1}, b_{4,1}, b_{5,1}$, and $b_{6,1}$, with 25% less I/O than the conventional repair for the $(6, 4)$ RS code (four full blocks). Note that LESS's repair I/O reductions apply to all data and parity blocks. For

any block, its sub-blocks must reside in one of the extended sub-strips. Thus, a single-block repair always retrieves six sub-blocks.

3.3 Construction

An (n, k, α) LESS stripe comprises n blocks B_i 's ($1 \leq i \leq n$), each with α sub-blocks $b_{i,j}$'s ($1 \leq j \leq \alpha$) (i.e., α sub-strips). LESS organizes the sub-blocks into $\alpha + 1$ extended sub-strips, each tolerating any $n - k$ sub-block failures, and constructs the extended sub-strips as follows.

Step 1 (Grouping blocks). We partition n blocks into $\alpha + 1$ block groups G_z 's ($1 \leq z \leq \alpha + 1$), whose numbers of blocks differ by at most one. The number of blocks in G_z ($|G_z|$) is:

$$|G_z| = \begin{cases} \lceil \frac{n}{\alpha+1} \rceil & \text{if } z \leq n \bmod (\alpha+1) \\ \lfloor \frac{n}{\alpha+1} \rfloor & \text{otherwise.} \end{cases} \quad (3)$$

In Figure 1, we divide the six blocks into three block groups: $G_1 = \{B_1, B_2\}$, $G_2 = \{B_3, B_4\}$, and $G_3 = \{B_5, B_6\}$.

Step 2 (Layering extended sub-strips). We organize $n\alpha$ sub-blocks into $\alpha + 1$ extended sub-strips X_z 's ($1 \leq z \leq \alpha + 1$). Let g_i be the index of the block group containing block B_i (i.e., if $B_i \in G_z$, then $g_i = z$). Each of the first α X_z 's (for $1 \leq z \leq \alpha$) includes all sub-blocks in G_z plus all sub-blocks in the z -th sub-stripe, while $X_{\alpha+1}$ includes all sub-blocks in $G_{\alpha+1}$ and all sub-blocks $b_{i,j}$'s where $g_i = j$. The number of sub-blocks in X_z ($|X_z|$) is:

$$|X_z| = n + (\alpha - 1)|G_z|, \quad (4)$$

where n accounts for the sub-blocks in a sub-stripe, and $(\alpha - 1)|G_z|$ accounts for the remaining sub-blocks in G_z that do not belong to the same sub-stripe. Each sub-block belongs to exactly two extended sub-strips. In particular, $X_{\alpha+1}$ includes the sub-blocks from the first α extended sub-strips, where each sub-block appears in exactly one of the first α extended sub-strips. In Figure 1, we organize the sub-blocks into three extended sub-strips (X_1 , X_2 and X_3), each of which contains eight sub-blocks.

Step 3 (Building MDS extended sub-strips). We construct X_z using an $(|X_z|, |X_z| - n + k)$ Vandermonde-based RS code to tolerate any $n - k$ sub-block failures. We select $n\alpha$ distinct coefficients $v_{i,j}$'s ($1 \leq i \leq n$ and $1 \leq j \leq \alpha$) from $\text{GF}(2^w)$, such that X_z must satisfy the parity-check equation:

$$\sum_{b_{i,j} \in X_z} b_{i,j} \mathbf{v}_{i,j} = \mathbf{0}, \quad (5)$$

where $\mathbf{v}_{i,j} = [1 \quad v_{i,j} \quad v_{i,j}^2 \quad \dots \quad v_{i,j}^{n-k-1}]^T$ is a Vandermonde column vector of length $n - k$. The linear independence of any $n - k$ Vandermonde column vectors ensures that X_z is an RS-coded stripe. In Figure 1, each extended sub-stripe is encoded using an $(8, 6)$ Vandermonde-based RS code.

MDS property. For a LESS stripe to tolerate any $n - k$ block failures, we must carefully select $n\alpha$ distinct coding coefficients $v_{i,j}$'s from a Galois Field. Theorem 1 shows that

such coefficients can always be found in a sufficient field size. The detailed proof is in the supplementary file.

Theorem 1. For (n, k, α) LESS, we can always find $n\alpha$ distinct coding coefficients $v_{i,j}$'s ($1 \leq i \leq n$ and $1 \leq j \leq \alpha$) in $\text{GF}(2^w)$ for MDS when $2^w \geq n\alpha + (n - k - 1) \binom{n-1}{k}$.

Theorem 1 provides a sufficient condition for the Galois Field size. In practice, for typical coding parameters, the required coefficients can be found in smaller fields ($\text{GF}(2^8)$ or $\text{GF}(2^{16})$). In our implementation, we can generate the $n\alpha$ coding coefficients using a primitive element p with a multiplicative approach:

$$v_{i,j} = p^{(h_i(\alpha+1)+g_i)\alpha+j}, \quad 1 \leq i \leq n \text{ and } 1 \leq j \leq \alpha, \quad (6)$$

where h_i ($1 \leq i \leq |G_z|$) is the position of B_i in its block group G_z (i.e., B_i is the h_i -th block in G_z). For example, in Figure 1, B_2 is the second block in G_1 , so $g_2 = 1$ and $h_2 = 2$. The primitive element is $p = 2$ for $(6, 4, 2)$ LESS, while $v_{2,1}$ and $v_{2,2}$ are 2^{15} and 2^{16} in $\text{GF}(2^8)$, respectively. We can find the feasible primitive elements based on brute-force search (i.e., checking linear independence by enumerating any possible sub-matrices of the parity-check matrix). Note that the search of feasible primitive elements is done only once for construction. In the supplementary file, we show how the parity-check matrix is formed and provide the feasible primitive elements for common (n, k, α) ($n - k \leq 4$ and $2 \leq \alpha \leq 4$).

3.4 Repair

LESS supports efficient repair of single-block failures within an extended sub-stripe and certain multi-block failures when the failed blocks reside in the same extended sub-stripe.

Single-block repair. For a failed block B_i ($1 \leq i \leq n$) in G_z ($1 \leq z \leq \alpha$), we repair B_i using sub-blocks within X_z , which contains all sub-blocks of G_z . Since X_z is an $(|X_z|, |X_z| - n + k)$ RS-coded stripe, it tolerates any $n - k \geq \alpha$ sub-block failures. The α failed sub-blocks can be reconstructed using the parity-check equation (Equation (5)) for X_z . To reduce accessed blocks, we prioritize transferring $|G_z|(\alpha - 1)$ available sub-blocks from G_z , followed by $|X_z| - n + k - |G_z|(\alpha - 1)$ available sub-blocks from other block groups.

Multi-block repair. When $\lfloor \frac{n-k}{\alpha} \rfloor \geq 2$, LESS can repair any $\lfloor \frac{n-k}{\alpha} \rfloor$ block failures within one extended sub-stripe if all failed blocks reside in the same block group, say G_z ($1 \leq z \leq \alpha + 1$). With all failed sub-blocks in X_z , since $\alpha \lfloor \frac{n-k}{\alpha} \rfloor \leq n - k$, the failed sub-blocks can be repaired within X_z . For other multi-block failure cases, LESS resorts to conventional repair (which retrieves k blocks). For example, in a $(14, 10, 2)$ LESS stripe (Figure 2), repairing two failed blocks B_1 and B_2 in G_1 (i.e., the sub-blocks $b_{1,1}$, $b_{1,2}$, $b_{2,1}$, and $b_{2,2}$ are in X_1) can leverage X_1 's $(19, 15)$ RS coding. This requires 15 sub-blocks, with 25% less repair I/O than the conventional repair of the $(14, 10)$ RS code.

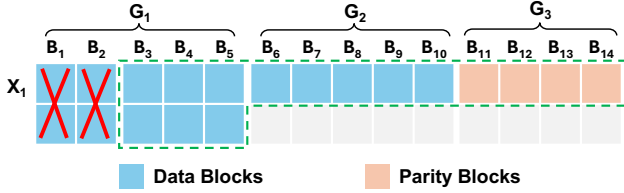


Figure 2: Two-block repair in $(n, k, \alpha) = (14, 10, 2)$ LESS.

3.5 I/O-Efficient Repairs in LESS

LESS achieves I/O-efficient repairs in three aspects: (i) repair I/O, (ii) I/O seeks, and (iii) balanced I/O reductions.

Repair I/O. For block B_i ($1 \leq i \leq n$) in G_z (recall $z = g_i$), We retrieve $|X_z| - n + k$ sub-blocks, as we repair the block with the $(|X_z|, |X_z| - n + k)$ RS code. From Equations (3) and (4), the repair I/O (in sub-blocks) of B_i is:

$$IO_i = \begin{cases} k + (\alpha - 1) \lceil \frac{n}{\alpha+1} \rceil & \text{if } g_i \leq n \bmod (\alpha + 1) \\ k + (\alpha - 1) \lfloor \frac{n}{\alpha+1} \rfloor & \text{otherwise.} \end{cases} \quad (7)$$

This is strictly less than the I/O for the conventional repair of an (n, k) RS code when $k > \lceil \frac{n}{3} \rceil$ (common in practice), as $IO_i \leq k + (\alpha - 1) \lceil \frac{n}{\alpha+1} \rceil \leq k + (\alpha - 1) \lceil \frac{n}{3} \rceil < k\alpha$.

I/O seeks. For a single-block failure, LESS limits I/O seeks to $k + \alpha - 1$. Repairing B_i in G_z requires $|X_z| - n + k$ sub-blocks from X_z , including: (i) the $\alpha(|G_z| - 1)$ contiguous sub-blocks from G_z , which can be retrieved with $|G_z| - 1$ seeks, and (ii) $|X_z| - n + k - \alpha(|G_z| - 1)$ sub-blocks from other blocks, each requiring one seek. The total number of I/O seeks is $(|G_z| - 1) + |X_z| - n + k - \alpha(|G_z| - 1) = k + \alpha - 1$, from Equations (3) and (4).

Balanced I/O reductions. The repair I/O is similar for the repair of any data or parity block in a stripe, and differs by at most $\alpha - 1$ sub-blocks from Equation (7). Also, the repair incurs *exactly one* I/O seek from each of the $k + \alpha - 1$ available blocks. Thus, LESS balances I/O reductions.

4 Evaluation

We evaluate LESS via numerical analysis and testbed experiments, and address two questions: (i) Does LESS's empirical repair performance conform to its theoretical improvements? (ii) How do system configurations affect repair performance?

4.1 Numerical Analysis

Exp#A1 (Single-block repair). We compare LESS against systematic MDS codes (RS, Clay, Hitchhiker, HashTag, and ET) in a single-block repair. We vary α from 2 to $n - k$ for HashTag, ET, and LESS. We measure the average, minimum, and maximum repair I/O (in blocks) and total number of I/O seeks when repairing each of the n blocks. We focus on $(n, k) = (14, 10)$, a default configuration in Facebook's f4 [21]. Table 2 shows that for $\alpha \geq 3$, LESS has less repair I/O than other codes (except Clay, which minimizes repair I/O) with

Codes	α	Repair I/O		# I/O seeks	
		Avg	Min/Max	Avg	Min/Max
RS	1	10.00	10.00 / 10.00	10.000	10 / 10
Clay	256	3.25	3.25 / 3.25	286.00	13 / 832
Hitchhiker	2	7.50	6.50 / 10.00	10.86	10 / 13
HashTag	2	7.07	5.50 / 10.00	10.71	10 / 11
	3	6.67	4.34 / 10.00	12.64	10 / 20
	4	6.04	4.00 / 10.00	12.14	10 / 13
ET	2	7.50	7.50 / 7.50	11.00	11 / 11
	3	7.14	6.67 / 7.67	13.43	12 / 14
	4	5.86	5.50 / 6.25	14.29	13 / 15
LESS	2	7.36	7.00 / 7.50	11.00	11 / 11
	3	5.71	5.33 / 6.00	12.00	12 / 12
	4	4.64	4.00 / 4.75	13.00	13 / 13

Table 2: Analysis for a single-block repair for $(14, 10)$.

Codes / (n, k)	Avg. Repair I/O / Avg. # of I/O seeks		
	(80, 76)	(100, 96)	(124, 120)
RS	76 / 76	96 / 96	120 / 120
LESS ($\alpha = 4$)	31 / 79	39 / 99	48.6 / 123

Table 3: Analysis for a single-block repair for wide-stripe codes.

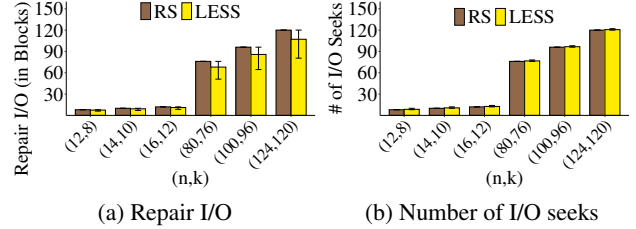


Figure 3: Analysis for two-block repairs for different (n, k) .

limited I/O seeks. For $\alpha = 4$, LESS reduces the average repair I/O of RS, Hitchhiker, HashTag ($\alpha = 4$), and ET ($\alpha = 4$) by 53.6%, 38.1%, 23.1%, and 20.7%, respectively, and reduces the average number of I/O seeks of Clay by 95.5%.

LESS also improves wide-stripe repairs. Table 3 shows the results (some parameters are studied in [9, 12]). For example, for $(124, 120)$, LESS ($\alpha = 4$) reduces the repair I/O of RS by 59.5% with limited extra I/O seeks.

Exp#A2 (Multi-block repair). We study two-block repair I/O and I/O seeks for LESS and RS codes (conventional repair) for different (n, k) 's. We average the results over all $\binom{n}{2}$ block failure combinations, with minimum/maximum error bars. Figure 3 shows the results. LESS reduces repair I/O for 27.3-32.8% of two-block failure cases, with limited I/O seek overhead. For example, for $(14, 10)$, LESS ($\alpha = 2$) reduces the average repair I/O of RS by 7.4% and improves repairs for 28.6% of cases. For wide stripes, say $(124, 120)$, LESS ($\alpha = 2$) reduces the average repair I/O of RS by 10.8% and improves repairs for 32.8% of cases.

4.2 Testbed Experiments

Implementation. We implement LESS on OpenEC (an erasure coding middleware) [19] atop Hadoop 3.3.4 HDFS [7] and use Jerasure [26] for coding operations. We also include RS, Clay, Hitchhiker, HashTag, and ET in our comparisons.

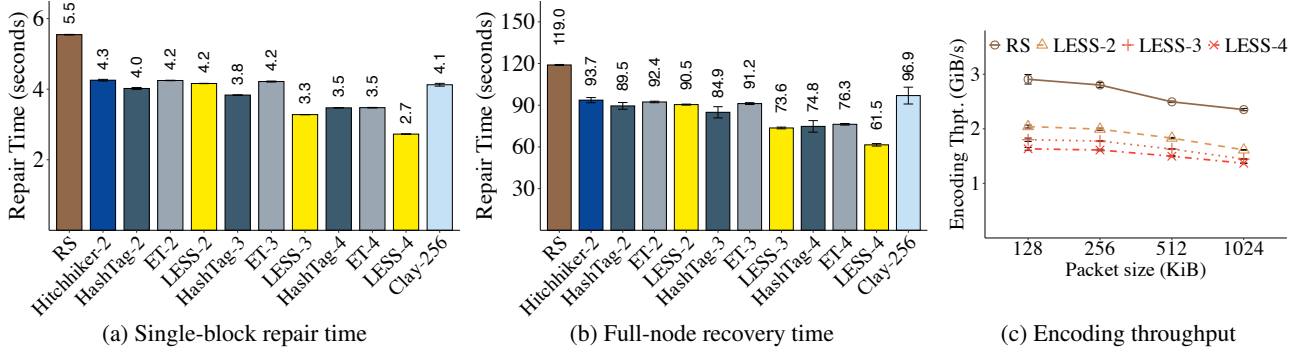


Figure 4: Repair performance for (14, 10). In figures (a) and (b), we put α next to each code (e.g., LESS-4 has $\alpha = 4$).

Our prototype adds 8.7 K LoC to OpenEC in C++.

HDFS uses a NameNode for storage management and multiple DataNodes for storage. HDFS organizes data in fixed-size blocks, each further partitioned into multiple *packets*. The packets at the same block offsets are encoded together, allowing pipelined coding operations across packets. Our prototype follows this packet-level pipelined implementation.

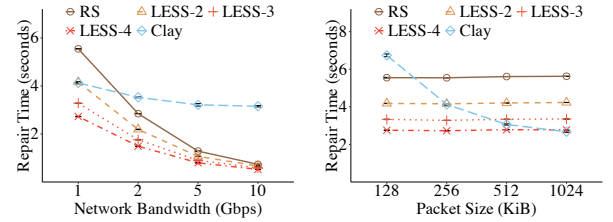
Methodology. We conduct testbed experiments in a 15-machine local cluster connected via a 10 Gbps Ethernet switch. Each machine is equipped with a quad-core 3.4 GHz Intel i5-7500 CPU, 16 GiB RAM, a 7200 RPM 1 TB SATA HDD, and Ubuntu 22.04. We use Wondershaper [11] to configure the network bandwidth of each machine. We use one machine for the NameNode and 14 machines for DataNodes. By default, we set $(n, k) = (14, 10)$, 64 MiB blocks, 256 KiB packets, and 1 Gbps network bandwidth as in prior work [19, 31, 35].

We evaluate the (i) *single-block repair time* (i.e., the time from issuing a repair request to a failed block until it is repaired, averaged over n blocks) and (ii) *full-node recovery time* (i.e., the total time to repair 20 blocks in a failed DataNode). We average results over 10 runs, with 95% confidence intervals based on the student's t-distribution.

Exp#B1 (Single-block repair). Figure 4(a) shows that LESS effectively reduces the single-block repair time due to the reductions in repair I/O and I/O seeks. For example, LESS ($\alpha = 4$) reduces the single-block repair times of RS, Hitchhiker, HashTag ($\alpha = 4$), ET ($\alpha = 4$), and Clay by 50.8%, 35.9%, 21.5%, 21.5%, and 33.9%, respectively.

Exp#B2 (Full-node recovery). Figure 4(b) shows that LESS also reduces the full-node recovery time. For example, the reductions of LESS ($\alpha = 4$) compared with RS, Hitchhiker, HashTag ($\alpha = 4$), ET ($\alpha = 4$), and Clay are 48.3%, 34.3%, 17.8%, 19.4%, and 36.6%, respectively.

Exp#B3 (Encoding throughput). We measure the encoding throughput (i.e., the amount of data encoded per second) on one machine. We construct a (14, 10) stripe in memory and put random bytes in the stripe for encoding. Figure 4(c) shows the results for packet sizes from 128 KiB to 1024 KiB. For 256 KiB packets, the encoding throughputs of RS and LESS ($\alpha = 4$) are 2.8 GiB/s and 1.6 GiB/s. Also, for wide stripes,



(a) Impact of network bandwidth (b) Impact of packet size

Figure 5: Impact of configurations on (14,10) single-block repair.

say (124, 120) and 256 KiB packets, the encoding throughput of RS and LESS ($\alpha = 4$) is 2.6 GiB/s and 1.1 GiB/s, respectively (not shown in the figure). LESS has lower encoding throughput than RS due to sub-packetization, yet its computational overhead remains limited compared to I/O operations.

Exp#B4 (Impact of network bandwidth). We study the impact of network bandwidth, varied from 1 Gbps to 10 Gbps via Wondershaper, on the single-block repair time. Figure 5(a) shows that as the network bandwidth increases, Clay suffers from significant I/O seek overhead, while LESS effectively reduces the single-block repair time due to small sub-packetization. For example, for 10 Gbps bandwidth, LESS ($\alpha = 4$) reduces the single-block repair times of RS and Clay by 28.6% and 83.3%, respectively. Overall, LESS maintains a low single-block repair time for varying network bandwidth.

Exp#B5 (Impact of packet size). We study the impact of packet size, varied from 128 KiB to 1024 KiB. Figure 5(b) shows that for small packet sizes, Clay incurs significant I/O overhead for processing a large number of sub-blocks, while LESS keeps stable repair performance. For example, for 128 KiB packets, LESS ($\alpha = 4$) reduces the single-block repair times of RS and Clay by 59.1% and 50.4%, respectively.

5 Conclusion

LESS is a family of erasure codes designed for I/O-efficient repairs by layering extended sub-stripes, so as to reduce both repair I/O and I/O seeks and ensure balanced reductions across blocks. It has several practical properties: MDS, general parameters, systematic, and configurable sub-packetization. Numerical analysis and testbed experiments show LESS's repair benefits over state-of-the-art repair-friendly codes.

References

- [1] S. B. Balaji, M. Nikhil Krishnan, Myna Vajha, Vinayak Ramkumar, Birenjith Sasidharan, and P. Vijay Kumar. Erasure coding for distributed storage: an overview. *Science China Information Sciences*, 61(100301):100301:1–100301:45, 2018.
- [2] S. B. Balaji, Myna Vajha, and P. Vijay Kumar. Lower bounds on the sub-packetization level of MSR codes and characterizing optimal-access MSR codes achieving the bound. *IEEE Trans. on Information Theory*, 68(10):6452–6471, 2022.
- [3] Brian Beach. Backblaze Vaults: Zettabyte-scale cloud storage architecture. <https://www.backblaze.com/blog/vault-cloud-storage-architecture/>, 2019.
- [4] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.
- [5] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.
- [6] Chuang Gan, Yuchong Hu, Leyan Zhao, Xin Zhao, Pengyu Gong, and Dan Feng. Revisiting network coding for warm blob storage. In *Proc. of USENIX FAST*, 2025.
- [7] Hadoop 3.3.4. <https://hadoop.apache.org/docs/r3.3.4/>, 2022.
- [8] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST*, 2012.
- [9] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for Wide-Stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, 2021.
- [10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In *Proc. of USENIX ATC*, 2012.
- [11] Bert Hubert, Jacco Geul, and Simon Séhier. Wonder Shaper. <https://github.com/magnific0/wonder-shaper>, 2012.
- [12] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical design considerations for wide locally recoverable codes (LRCs). In *Proc. of USENIX FAST*, 2023.
- [13] Osama Khan, Randal Burns, James Plank, and William Pierce. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.
- [14] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.
- [15] Katina Kravevska and Danilo Gligoroski. An explicit construction of systematic MDS codes with small sub-packetization for all-node repair. In *Proc. of IEEE DASC/PiCom/DataCom/CyberSciTech*, 2018.
- [16] Katina Kravevska, Danilo Gligoroski, Rune E. Jensen, and Harald Øverby. HashTag erasure codes: From theory to practice. *IEEE Trans. on Big Data*, 4(4):516–529, 2018.
- [17] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.
- [18] Xiaolu Li, Keyun Cheng, Kaicheng Tang, Patrick P. C. Lee, Yuchong Hu, Dan Feng, Jie Li, and Ting-Yi Wu. ParaRC: Embracing sub-packetization for repair parallelization in MSR-coded storage. In *Proc. of USENIX FAST*, 2023.
- [19] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, 2019.
- [20] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.
- [21] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook’s warm BLOB storage system. In *Proc. of USENIX OSDI*, 2014.
- [22] Lluís Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandić. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.
- [23] Andreas-Joachim Peters, Michal Kamil Simon, and Elvin Alin Sindrilaru. Erasure coding for production in the EOS open storage system. In *Proc. of CHEP*, 2019.
- [24] James S. Plank and Cheng Huang. Tutorial: Erasure coding for storage applications. <http://web.eecs.utk.edu/~jplank/plank/papers/FAST-2013-Tutorial.html>, 2013.
- [25] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O’Hearn. A performance

- evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX FAST*, 2009.
- [26] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - version 2.0. Technical Report CS-08-627, 2014.
- [27] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.
- [28] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.
- [29] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.
- [30] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
- [31] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: novel erasure codes for big data. In *Proc. of VLDB Endowment*, 2013.
- [32] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *Proc. of ACM SOSP*, 2021.
- [33] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick P. C. Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A survey of the past, present, and future of erasure coding for storage systems. *ACM Trans. on Storage*, 21(1):1–39, 2025.
- [34] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc. of IEEE MSST*, 2010.
- [35] Kaicheng Tang, Keyun Cheng, Helen H. W. Chan, Xiaolu Li, Patrick P. C. Lee, Yuchong Hu, Jie Li, and Ting-Yi Wu. Balancing repair bandwidth and sub-packetization in erasure-coded storage via elastic transformation. In *Proc. of IEEE INFOCOM*, 2023.
- [36] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayana-murthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.
- [37] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, 2006.
- [38] Yunfeng Zhu, Patrick P. C. Lee, Yinlong Xu, Yuchong Hu, and Liping Xiang. On the speedup of recovery in large-scale erasure-coded storage systems. *IEEE Trans. on Parallel and Distributed Systems*, 25(7):1830–1840, 2014.

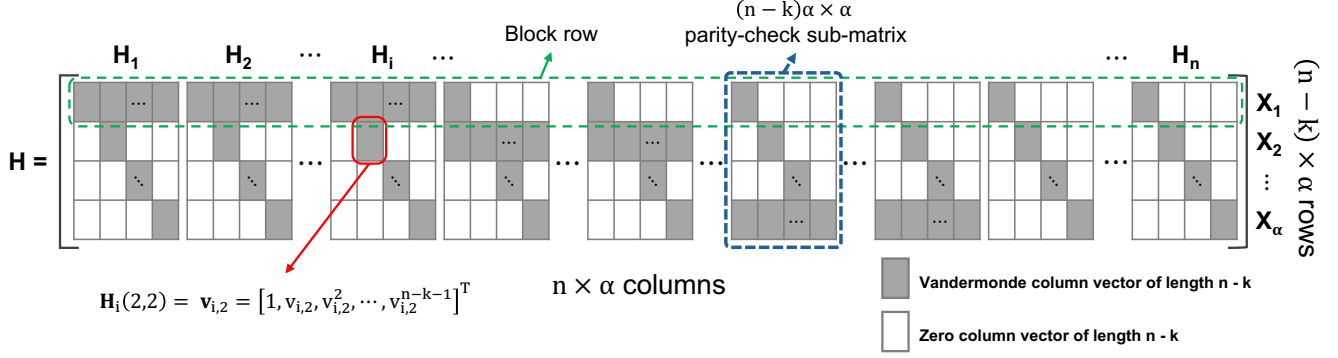


Figure 6: Parity-check matrix of an (n, k, α) LESS stripe.

A Proof of MDS Property

We prove Theorem 1 that LESS is MDS for a sufficiently large field size. Our proof is based on the parity-check matrix of LESS.

Parity-check matrix of LESS. The parity-check matrix (\mathbf{H}) of an (n, k, α) LESS stripe is an $(n-k)\alpha \times n\alpha$ matrix. We view \mathbf{H} as an $\alpha \times n\alpha$ block matrix, where each entry is a column vector of length $n-k$. We partition \mathbf{H} into n parity-check sub-matrices as $\mathbf{H} = [\mathbf{H}_1, \mathbf{H}_2, \dots, \mathbf{H}_n]$, where each \mathbf{H}_i (where $1 \leq i \leq n$) is an $\alpha \times \alpha$ block matrix. We define a block row as $n\alpha$ entries in a row of \mathbf{H} . The entry of \mathbf{H}_i on the z -th block row and the j -th column (where $1 \leq z \leq \alpha$ and $1 \leq j \leq \alpha$) is given by

$$\mathbf{H}_i(z, j) = \begin{cases} \mathbf{v}_{i,j} & \text{if } b_{i,j} \in X_z \\ \mathbf{0} & \text{otherwise.} \end{cases} \quad (8)$$

Figure 6 shows the parity-check matrix of LESS. We write block $B_i = [b_{i,1}, b_{i,2}, \dots, b_{i,\alpha}]^T$ as a column vector of the sub-blocks. From the parity-check matrix, the α parity-check equations of the first α extended sub-stripes can be written as

$$[\mathbf{H}_1, \mathbf{H}_2, \dots, \mathbf{H}_n][B_1^T, B_2^T, \dots, B_n^T]^T = \sum_{i=1}^n \mathbf{H}_i B_i = \mathbf{0}, \quad (9)$$

Based on Equations (5) and (9), the z -th block row can be written as the parity-check equation of the extended sub-stripe X_z as follows:

$$\begin{aligned} & \sum_{i=1}^n [\mathbf{H}_i(z, 1), \mathbf{H}_i(z, 2), \dots, \mathbf{H}_i(z, \alpha)] B_i \\ &= \sum_{i=1}^n \left[\sum_{j=1}^{\alpha} (\mathbf{H}_i(z, j) b_{i,j}) \right] = \sum_{b_{i,j} \in X_z} b_{i,j} \mathbf{v}_{i,j} = \mathbf{0}. \end{aligned}$$

Figure 7 shows the parity-check matrix of $(6, 4, 2)$ LESS. The 4×12 parity-check matrix \mathbf{H} comprises two block rows and six parity-check sub-matrices. The first and second block rows can be derived from the parity-check equations of X_1 and X_2 , respectively.

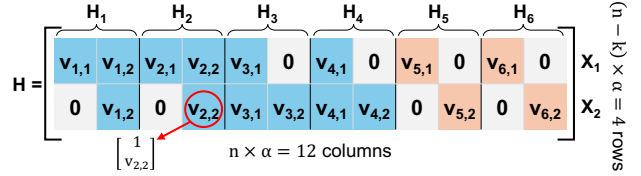


Figure 7: Parity-check matrix of a $(6, 4, 2)$ LESS stripe.

$n-k$	α	n	Galois Fields	p
2	2	$n \leq 127$	$\text{GF}(2^8)$	2
3	2	$n \leq 44$	$\text{GF}(2^8)$	50
		$n \leq 127$	$\text{GF}(2^{16})$	2
	3	$n \leq 40$	$\text{GF}(2^8)$	14
		$n \leq 127$	$\text{GF}(2^{16})$	2
4	2	$n \leq 23$	$\text{GF}(2^8)$	6
		$n \leq 127$	$\text{GF}(2^{16})$	46
	3	$n \leq 17$	$\text{GF}(2^8)$	2
		$n \leq 127$	$\text{GF}(2^{16})$	1362
	4	$n \leq 16$	$\text{GF}(2^8)$	14
		$n \leq 127$	$\text{GF}(2^{16})$	635

Table 4: Feasible primitive elements for commonly used coding parameters.

Proof of Theorem 1. To ensure LESS is MDS, we can verify the parity-check matrix of LESS, such that any $n-k$ out of n parity sub-matrices can form an $(n-k)\alpha \times (n-k)\alpha$ square invertible matrix. Recall that any invertible square matrix has a non-zero determinant. To ensure that all possible square matrices from the parity-check matrix are invertible, we can multiply the determinants of the square matrices and ensure that the product is non-zero.

There are $\binom{n}{n-k}$ cases for selecting $n-k$ matrices out of the n parity-check sub-matrices to form a square matrix. The determinant of each square matrix can be viewed as a polynomial over the coding coefficients $v_{i,j}$ (where $1 \leq i \leq n$ and $1 \leq j \leq \alpha$). Each $v_{i,j}$ appears in only one parity-check sub-matrix (i.e., \mathbf{H}_i), and there are $\binom{n-1}{k}$ possible square matrices

that contain \mathbf{H}_i . Thus, the degree of $v_{i,j}$ in the determinant of such square matrix is $n - k - 1$. By multiplying the determinants of all square matrices, the degree of each $v_{i,j}$ is $(n - k - 1) \binom{n-1}{k}$. We can choose $n\alpha$ distinct coding coefficients in $\text{GF}(2^w)$ to ensure the product of determinants is non-zero when $2^w \geq n\alpha + (n - k - 1) \binom{n-1}{k}$, which can be proven from Noga Alon's Combinatorial Nullstellensatz. \square

Table 4 shows the feasible primitive elements for commonly used coding parameters $n \leq 127$ and $n - k \leq 4$ in $\text{GF}(2^8)$ and $\text{GF}(2^{16})$ based on brute-force search.