

DuraShare: BIP39-native secret sharing with full manual fallback

Jeroen van de Graaf¹ and Renato Schiavinato Lopez²

¹ DCC/UFGM

jvdg@dcc.ufmg.br

² Grifortis

renato.lopez@grifortis.com

Abstract. BIP39 is a Bitcoin standard that turns a wallet’s random seed of size between 128 and 256 bits into a human-readable sequence of 12 to 24 words, taken from a fixed 2048-word list. Each word corresponds to an index in that list. DuraShare allows the holder of this sequence to split it among n *Sharekeepers*, each of whom receives one *share*. Any subset of k Sharekeepers can combine their shares and reconstruct the original sequence, whereas $k - 1$ colluding Sharekeepers obtain no (Shannon) information.

For everyday convenience we adopt a software-assisted model, while ensuring that a full manual fallback remains feasible for recovery, requiring nothing beyond paper, a calculator, and specific instructions. This matters because Sharekeepers may be family members or lawyers unfamiliar with crypto-technology, called upon only decades later, when no specific vendor, software, or hardware can be assumed to still exist.

DuraShare combines three existing, well-established cryptographic techniques: (1) *Shamir Secret Sharing* over $\mathbb{F}_{2053}$, the smallest prime larger than 2048; (2) *2D error detection*, applying row and column checksums to each share to protect the data against transcription errors and physical damage; and (3) a simple *affine inner-product MAC* over $\mathbb{F}_{2053}$ to authenticate each share, preventing a Sharekeeper from tampering without being detected.

Keywords: BIP39 · Shamir secret sharing · manual key backup · crypto-wallet

1 Introduction

1.1 Motivation

A Bitcoin address is nothing but the hash of a public key. The owner of the matching private key can sign a transaction to spend the assets it protects. The same holds for Ethereum and other cryptocurrencies. Owners can opt to outsource key storage and management to brokers or exchanges, but that leaves them vulnerable to these exchanges being hacked, as happened in the infamous Mt. Gox hack. Instead, to stay in full control, the owner can take key storage and management into his own hands.

To do so, the owner needs a crypto-wallet, a hardware device or software application that helps generate and manage these private keys. Typically, a wallet takes a random seed of 128, 192, or 256 bits and applies repeated hashing to derive a hierarchical tree of private keys, which are then used to sign transactions. The secrecy of this random seed is of vital importance. If it leaks, another person could steal the assets. But recoverability is also of vital importance. If the seed gets lost, the assets it protects are lost too.

The BIP39 standard encodes a wallet’s random seed of 128 to 256 bits as a sequence of 12, 18 or 24 numbers in $\{1, \dots, 2048\}$, obtained by splitting the seed into consecutive 11-bit groups [8]. Each index is displayed to the user as a word from a fixed 2048-word list, chosen so that the first four letters uniquely identify each word [7]. Even though initially designed for Bitcoin, BIP39 has been widely adopted by other crypto-currencies and wallets. For instance, MetaMask also uses BIP39.

A wallet generates a random seed, turns it into a sequence of words from this list, and urges the user to write the words down on paper. As a backup, the owner could make copies of this sequence and hide them in various places. But each copy is just as powerful as the original: anyone who finds one can spend all the funds, and more copies mean more chances of that happening. This is why secret sharing is desirable. It lets the owner split the seed into several shares, so that no small group of them reveals anything about the seed, and only a large enough group can recover it.

Secret sharing is useful for making a backup, but we are particularly concerned with a harder case: the owner is no longer alive. Crypto-assets can be worth a great deal, and their recovery must be protected even after the owner dies. An added complication is that the heirs may be completely averse to technology. Clearly we are talking about long-term scenarios, spanning decades, so we cannot even be sure that the hardware device or software application needed for recovery still exists.

This is why we advocate the following model: *the sharing ceremony and the recovery ceremony are software-assisted (using an air-gapped computer), but manual recovery always remains available as a fallback, requiring nothing beyond paper, a calculator, and this specification.*

1.2 Contributions of this paper

This paper makes the following contributions:

- We instantiate Shamir Secret Sharing over $\mathbb{F}_{2053}$, the smallest prime larger than the 2048-word BIP39 alphabet, and justify this choice against the alternatives on the specific grounds of manual arithmetic feasibility using a basic calculator.
- We give an explicit, rejection-sampling procedure for generating uniform values in $\mathbb{F}_{2053}$ from ordinary playing cards.
- We eliminate the computation of modular inverses during manual recovery by precomputing the Lagrange coefficients.

- We propose a row/column/global-integrity checksum structure, and prove that it achieves minimum distance $d_{\min} = 4$, so that every error of weight up to 3 is detected with certainty.
- By adding a second, RS based, parity equation to the above row-column structure, we can raise the minimum distance to $d_{\min} = 9$. This detects up to 8 errors with certainty. Decoding each row or column independently by hand guarantees correction of any pattern with at most one error per row and per column, while device-assisted syndrome decoding can correct up to 4 errors anywhere.
- More generally, we show it is possible to correct a single error entirely *by hand* for any $(n + 2, n)$ Reed-Solomon code with two check symbols, over any prime field $\mathbb{F}_p$. This may be of separate interest.
- Based on the Carter-Wegman construction, we introduce an affine inner-product MAC and prove it secure. This enables the optional auditing of a single share in isolation, without recovering the secret or combining it with any other share.
- The row/column checksum and RS constructions do not depend on any specific grid shape: any factorization $\ell = r \times c$ works, with the checksum layer generalizing unchanged and the RS layer requiring only that each row and column admit a distance-3 RS component code. We illustrate this with 3×4 , 5×4 (zero-padded), and 6×4 arrays for $\ell = 12, 18, 24$, but the construction accommodates other factorizations or padding choices as needed.

A reference implementation, thoroughly tested and published in good faith, but not independently audited, is available along with test vectors at <https://github.com/GRIFORTIS/durashare>.

1.3 Comparison with other work and paper outline

As far as we know, no existing systems combines all the properties that we sought to combine in DuraShare:

1. k -out-of- n secret sharing for $k < n$,
2. arithmetic simple enough to be feasible by hand,
3. simple error detection, and
4. auditing of a single share in isolation.

SeedXOR [3] also aims computer-free recovery, splitting raw seed entropy via bitwise XOR rather than Shamir Secret Sharing. In practice, this means converting each word to a triple of hex digits, then computing the XOR of each digit position by consulting a 16×16 table. This arithmetic is extremely error-prone for humans. Moreover, SeedXOR is strictly n -of- n , with no threshold at all. So losing a share or erring even a single bit destroys the secret permanently, with essentially no error detection beyond each part’s own generic BIP39 checksum to catch a mistake. We suggest an improvement on SeedXOR in Appendix D.

Codex32 [9] uses a distance-9 BCH code, theoretically correcting up to four character substitutions. In practice, however, only single-error correction is manually supported, via a precomputed lookup table. The authors state plainly that

general multi-error correction requires Berlekamp–Massey decoding and Forney’s algorithm over $\mathbb{F}_{1024}$, which they had not, as of their own 2023 writing, found a way to perform with volvelles or worksheets [12].

SLIP39 [10] is SatoshiLabs’ Shamir-based standard for Trezor devices. It shares DuraShare’s core primitive but diverges in field choice and scope. It splits the secret over $\mathbb{F}_{256}$, a binary extension field using the AES/Rijndael irreducible polynomial, introducing exactly the kind of polynomial arithmetic we wanted to avoid. More fundamentally, SLIP39 uses a different key derivation function; the same entropy does not produce the same wallet as BIP39.

SSKR [6] is a BIP39-compatible alternative based on SLIP-39. Its optional Bytewords CRC-32 detects damaged encodings, but not malicious replacements with recomputed checksums. Its four-byte HMAC-SHA256 digest is checked only after threshold reconstruction, so it cannot authenticate an individual share; with $k - 1$ shares, it also provides an offline candidate test with false-positive probability 2^{-32} , reducing effective entropy by up to 32 bits [1].

Paper outline The organization of this paper is straightforward: Section 2 contains a high-level description of our method, while Sections 3 and 4 present the sharing ceremony, and the recovery ceremony respectively. Section 5 closes with some final remarks and conclusions.

2 High-level description of our method

2.1 Parties

DuraShare involves two roles. The *Principal* holds a secret sequence of $\ell \in \{12, 18, 24\}$ indices in $\{1, \dots, 2048\}$ (the index sequence underlying a BIP39 mnemonic) and wishes to back it up such that it remains recoverable even if the Principal becomes unavailable or loses all local copies. The Principal computes, from this secret sequence, a sequence of ℓ field elements in $\mathbb{F}_{2053}$ for each of n Sharekeepers, and distributes them at setup. A *Sharekeeper* holds one *share*: public metadata plus this sequence of field elements.

No single Sharekeeper, nor any coalition of fewer than k Sharekeepers, learns anything about the Principal’s secret sequence from their shares alone. But any coalition of k Sharekeepers, pooling their shares, can reconstruct it exactly. Note that the Principal and the Sharekeepers may be the same physical person acting in different capacities at different times (e.g., a single individual splitting their own mnemonic and distributing shares to trusted locations) or distinct people, as in an inheritance or professionally-assisted setup.

2.2 BIP39

BIP39 encodes a wallet’s random seed of 128 to 256 bits as a sequence of $\ell \in \{12, 18, 24\}$ indices in $\{1, \dots, 2048\}$, obtained by splitting the seed into consecutive 11-bit groups. When a program is used, each index is displayed to the user as a word from a fixed 2048-word list, but our cryptographic method

operates directly on the sequence of indices. So the correspondence between the words and indices is assured by the program.

There is, in fact, a mismatch between the 11-bit index groups and the raw seed sizes of 128, 192, and 256 bits. In the case of a 12-word mnemonic, the 12 indices correspond to $12 \times 11 = 132$ bits, four more than the 128 bits of entropy. These extra four bits are not arbitrary: they form a truncated hash: the 128-bit entropy is hashed with SHA-256, and the first 4 bits of that hash are appended before the combined 132-bit string is split into 11-bit groups. Likewise, 18 words carry a 6-bit reduced hash, and 24 words carry 8 bits.

2.3 Shamir secret sharing

Shamir Secret Sharing [11] realizes a (k, n) -threshold access structure over a finite field $\mathbb{F}_p$. A secret $w \in \mathbb{F}_p$ is split into n shares such that any k shares determine w uniquely, while any $k - 1$ shares are information-theoretically independent of w . The construction rests on a single fact from polynomial interpolation: k distinct points $(x_i, y_i) \in \mathbb{F}_p^2$ determine a unique polynomial of degree at most $k - 1$ passing through them, and no fewer.

Splitting. To share the secret w , the Principal samples coefficients $a_1, \dots, a_{k-1}$ uniformly and independently from $\mathbb{F}_p$, defines

$$f(X) = w + a_1X + a_2X^2 + \dots + a_{k-1}X^{k-1} \in \mathbb{F}_p[X],$$

and issues share i as the evaluation $(x_i, f(x_i))$, for n distinct public, nonzero $x_1, \dots, x_n \in \mathbb{F}_p$. Since f is a uniformly random polynomial of degree at most $k - 1$ subject only to $f(0) = w$, any $k - 1$ evaluations are jointly uniform over $\mathbb{F}_p^{k-1}$ independent of w : perfect secrecy holds against any coalition with less than k shares.

Recovery. Given k shares $(x_1, y_1), \dots, (x_k, y_k)$, Lagrange interpolation recovers f , and hence $w = f(0)$, as

$$w = \sum_{j=1}^k y_j \prod_{\substack{m=1 \\ m \neq j}}^k \frac{x_m}{x_m - x_j} \pmod{p}.$$

The Lagrange coefficients $\lambda_j = \prod_{m \neq j} x_m / (x_m - x_j)$ depend only on the x_i 's, so they can be precomputed. So recovery is a fixed public linear combination of the k share values. This observation is crucial, since it eliminates the need for computing inverses in $\mathbb{F}_p$, thus simplifying manual recovery. See the text before Step R2.

2.4 Choosing the Field Size

It is Shamir Secret Sharing that primarily determines the choice of field size $q = p^e$. Shamir reconstruction needs genuine field structure, and this choice then determines how easy or hard the computations are for a human throughout

the protocol. Mathematically, $\mathbb{F}_{2^{11}}$ is the natural candidate, since $2^{11} = 2048$ matches the BIP39 wordlist length exactly. However, binary-field arithmetic is notoriously difficult for humans: multiplication in $\mathbb{F}_{2^{11}}$ is **not** multiplication modulo 2048, but polynomial multiplication followed by reduction modulo an irreducible polynomial. Computers handle this without difficulty; humans do not.

For a system designed for manual recovery, the field arithmetic must be executable with pencil, paper, and a simple calculator. Prime fields $\mathbb{F}_p$ of moderate size, with ordinary integer arithmetic modulo p , are far more accessible to humans than binary fields, which require tables (or *volvelles*, see [9]), polynomial operations, or specialized software.

The logical choice is $p = 2053$, the smallest prime larger than 2048: it represents every word unambiguously, and ordinary calculators have sufficient precision to perform computations modulo 2053. This means that the integer values 0, 2049, 2050, 2051 and 2052 may appear as intermediate values in our calculations.

We did consider using smaller prime bases with multi-digit encoding: $q = 11$, using $m = 4$ base-11 digits per word (11 is the first prime exceeding 10), and $q = 13$, using $m = 3$ digits (since $13^3 = 2197 > 2048$). Arithmetic per digit is simpler in both cases, potentially doable without a calculator, but multi-digit encoding increases the number of manual operations per word and hence the error surface, so we rejected both in favor of the single-digit $p = 2053$ encoding.

2.5 Generating randomness

For the coefficients $a_1, \dots, a_{k-1}$ of the polynomial f , the Principal needs to generate random values modulo 2053. The authentication step described later also needs randomness, for κ and μ .

We use a reduced, 27-card deck and a rejection-sampling scheme based on $7^3 \cdot 6 = 2058 = 2053 + 5$. Each integer $0 \leq i \leq 2057$ is represented as $i = d_3 \cdot 294 + d_2 \cdot 42 + d_1 \cdot 6 + d_0$, with $d_3, d_2, d_1 \in \{0, \dots, 6\}$ and $d_0 \in \{0, \dots, 5\}$, mapped onto card suits and ranks so that a single draw of four cards yields one such representation. Any draw representing 2053 through 2057 is rejected and a fresh attempt begins, which happens with probability $5/2058$, less than 1 in 400. The first card obtained from each suit is uniform within that suit and independent of the other three, so the procedure gives 2058 equally likely outcomes, while rejecting the five invalid ones leaves a uniform value in $\mathbb{F}_{2053}$. For details see Appendix A.

2.6 Error detection through 2D parity check

We stress that device-assisted error correction should never apply a correction automatically. Instead, it should report the candidate position and magnitude and require operator confirmation, allowing the opportunity to detect a mistake.

Each share consists of ℓ field elements, one for every index of the underlying secret sequence. Manually copying a sequence of many numerical values invites exactly the kind of error humans routinely make when copying digits by hand:

substituting a visually similar digit ($1 \leftrightarrow 7$, $4 \leftrightarrow 9$), digit transposition ($35 \leftrightarrow 53$), or slipping on a single digit within a longer number ($2445 \rightarrow 2455$).

Arranging the ℓ values into a two-dimensional array, rather than a single row, gives us two independent opportunities to catch such an error, once via a row checksum and once via a column checksum, thus localizing a mistake to a single cell instead of merely flagging that *some* value among ℓ is wrong. We present an example with 3 rows and 4 columns.

$$\begin{array}{cccc|c} s_1 & s_2 & s_3 & s_4 & c_1 \\ s_5 & s_6 & s_7 & s_8 & c_2 \\ s_9 & s_{10} & s_{11} & s_{12} & c_3 \\ \hline e_1 & e_2 & e_3 & e_4 & g \end{array}$$

The data symbols are protected by row parity checks, column parity checks, and a global parity check that is equal to the sum of row checks or column checks. The code's minimum distance is $d_{\min} = 4$. This can be seen by pointing out that it is a product code of two single parity check codes, each with minimum distance two [4]. Alternatively, one can see that an error pattern is undetectable if and only if every row sum and every column sum is zero. The minimal nonzero pattern satisfying these constraints is a 2×2 rectangle $\begin{bmatrix} +\epsilon & -\epsilon \\ -\epsilon & +\epsilon \end{bmatrix}$ placed at the intersection of any two distinct rows and any two distinct columns. This pattern has Hamming weight 4 providing an upper bound on $d_{\min}$. And enumerating all the possible error layouts, one can easily show that no weight-3 pattern can be undetectable thus providing a lower bound.

Theorem 1 (Minimum Distance). *The array checksum code has minimum distance $d_{\min} = 4$: no error pattern of weight 1, 2, or 3 is undetectable, and the weight-4 rectangle pattern above shows this bound is tight.*

Theorem 2 (Single-Line Correction). *The array checksum code can correct any number of errors, provided they are confined to a single row or a single column.*

Proof. Suppose the errors are confined to one row (column). Then use the checksum in each column (row) to correct each error. $\square$

The global integrity check (GIC) $g = \sum_{i=1}^{12} s_i$ is algebraically redundant, as it equals both the sum of the row checks and the sum of the column checks:

$$g = c_1 + c_2 + c_3 = e_1 + e_2 + e_3 + e_4.$$

It therefore imposes no additional constraints on the code and does not alter the minimum distance. However, it allows a person to literally double-check his summations.

2.7 Reed–Solomon codes.

Based on polynomials, Reed–Solomon (RS) codes exist for any prime field, not just for binary fields. We illustrate the construction with rows of 6 words, protected by an RS(8,6) code over $\mathbb{F}_{2053}$ with 6 data symbols and 2 parity symbols. This code is capable of detecting 2 errors per row, or correcting one error per row, *manually*! We pick 6 as the largest row length actually used elsewhere in this paper (see Appendix D); the same construction with shorter rows is simply a truncation of the equations below, dropping the terms for the unused positions.

For example, for data symbols $s_1, \dots, s_6$, the parity symbols are

$$p_1 = \sum_{i=1}^6 i \cdot s_i \pmod{2053}, \quad p_2 = \sum_{i=1}^6 i^2 \cdot s_i \pmod{2053}.$$

Note that p_1 is simply the sum of each row, as explained in the previous section. The second parity equation adds a powerful error-correction capability.

For a received row $(\hat{s}_1, \dots, \hat{s}_6, \hat{p}_1, \hat{p}_2)$, the syndromes are

$$S_1 = \sum_{i=1}^6 i \cdot \hat{s}_i - \hat{p}_1 \pmod{2053}, \quad S_2 = \sum_{i=1}^6 i^2 \cdot \hat{s}_i - \hat{p}_2 \pmod{2053}.$$

If $S_1 = S_2 = 0$, there is no error. For a single error, its position and magnitude can be found from (S_1, S_2) . If only one of the two syndromes is zero, the error lies in the corresponding parity symbol; its magnitude is the negative of the nonzero syndrome. If both syndromes are nonzero, the error lies in some data position $i \in \{1, \dots, 6\}$. To find it, check $i \cdot S_1 \stackrel{?}{=} S_2$ for $i = 1, \dots, 6$ by repeatedly adding S_1 to itself. Once i is found, the error magnitude is $e = S_1 \cdot i^{-1} \pmod{2053}$. The inverses of 2, 3, 4, 5, and 6 modulo 2053 can be found by consulting a small table. See Appendix C for details.

2.8 Two-dimensional Reed–Solomon product code

This construction can be generalized to a two-dimensional array as well: a 3×4 array for 12 words, a 5×4 array for 18 words (with two padding cells fixed to zero), and a 6×4 array for 24 words (or 5×5 with padding). In every case, each row and column is protected by an RS code with two check symbols. The resulting 2D code has minimum distance $d_{\min} = 9$ in all three cases, detecting up to 8 errors with certainty, and, given software-assisted syndrome decoding, correcting up to 4 errors. Manually, using the per-line syndrome decoding explained above, each configuration still guarantees correction of any pattern with at most one error per row and at most one error per column.

2.9 Message Authentication Codes

Traditionally, a Message Authentication Code, MAC for short, is used to avoid some adversary modifying an encrypted message sent from A to B. The message

m is digested with a random function H , which has the property that no two messages are likely to result in the same MAC. Forging a MAC means guessing right despite the random choice of H , known to A and B but not to the adversary. In our scenario, the Principal uses it to detect tampering by a Sharekeeper.

This function H is chosen at random from a larger collection. A universal hash family $\mathcal{H}$ can be chosen so that every function H is easy to specify and to evaluate. A simple example is the *inner-product* universal hash family: fix a field $\mathbb{F}_p$, write the message as a vector $\mathbf{x} = (1, x_1, \dots, x_N) \in \mathbb{F}_p^{N+1}$ with the leading coordinate fixed to 1. Let each member of the family be indexed by a key $\boldsymbol{\kappa} = (\kappa_0, \dots, \kappa_N) \in \mathbb{F}_p^{N+1}$, chosen uniformly at random, via

$$H_{\boldsymbol{\kappa}}(\mathbf{x}) = \sum_{i=0}^N \kappa_i x_i \pmod{p}.$$

Evaluating $H_{\boldsymbol{\kappa}}$ requires only N multiplications and additions modulo p and can be performed with a basic calculator. Universal hashing was introduced by Carter and Wegman [2], while strongly universal hash functions and their application to MACs was presented in their follow-up paper [14].

3 The Sharing Ceremony

Assume the Principal has a mnemonic of $\ell \in \{12, 18, 24\}$ words corresponding to a sequence of indices $w_1, \dots, w_\ell \in \{1, \dots, 2048\}$. Our method secret-shares this sequence among n Sharekeepers, such that any subgroup of k Sharekeepers can reconstruct the original index sequence back.

Step S1: Generating the Base Shares

Input: An ℓ -word mnemonic $\mathbf{w} = (w_1, \dots, w_\ell)$ with each $w_i \in \{1, \dots, 2048\}$.

Output: n raw shares $\mathbf{s}^{(j)} = (s_1^{(j)}, \dots, s_\ell^{(j)})$ for $j \in \{1, \dots, n\}$, each $s_i^{(j)} \in \mathbb{F}_{2053}$.

Procedure:

1. For $i \in \{1, \dots, \ell\}$, construct a Shamir polynomial of degree at most $k - 1$: $f_i(x) = w_i + a_{i,1}x + \dots + a_{i,k-1}x^{k-1} \pmod{2053}$, with $a_{i,1}, \dots, a_{i,k-1}$ chosen independently and uniformly at random from $\mathbb{F}_{2053}$.
2. For $j \in \{1, \dots, n\}$: evaluate $s_i^{(j)} = f_i(j)$ for each $i \in \{1, \dots, \ell\}$, and assemble $\mathbf{s}^{(j)} = (s_1^{(j)}, \dots, s_\ell^{(j)})$. □

Theorem 3. *Any collection of k distinct shares uniquely determines the original mnemonic $\mathbf{w}$, while any collection of $k - 1$ shares reveals no information about it.*

This follows directly from Shamir's secret sharing scheme, as discussed in Section 2.3.

Step S2: Expanding a Share with Checksums

Input: A raw share $\mathbf{s} = (s_1, \dots, s_\ell) \in \mathbb{F}_{2053}^\ell$ (the share index j is dropped for notational simplicity; the procedure is identical for every share).

Output: The expanded share $\mathbf{t} = (\mathbf{s}, \mathbf{c})$, where $\mathbf{c}$ collects the checksums $c_1, \dots, c_r, d_1, \dots, d_r, e_1, \dots, e_4, f_1, \dots, f_4$, and the four corner values g_1, g_2, g_3, g_4 .

Procedure: Arrange $\mathbf{s}$ into $r = \lceil \ell/4 \rceil$ rows of 4 columns, padding the final row with zeros if ℓ is not a multiple of 4. All computations are modulo 2053.

For each row i , compute a plain checksum c_i (including the row number i as an additive tag, to catch a wrong-row error) and an RS-style checksum d_i (all computations modulo 2053):

$$c_i = \left(\sum_{m=1}^4 s_{4i-4+m} \right) + i, \quad d_i = \sum_{m=1}^4 m^2 \cdot s_{4i-4+m}.$$

For each column m , compute the analogous plain checksum e_m (including a column tag from $\{100, 200, 300, 400\}$, to catch a wrong-column error) and RS-style checksum f_m :

$$e_1, \dots, e_4 = \left(\sum_{i=1}^r s_{4i-4+m} \right) + 100m, \quad f_m = \sum_{i=1}^r i^2 \cdot s_{4i-4+m}.$$

The row checksums (c_i, d_i) and column checksums (e_m, f_m) are themselves protected, by applying the same column code to the c and d columns:

$$g_1 = \sum_{i=1}^r c_i, \quad g_2 = \sum_{i=1}^r i^2 \cdot c_i, \quad g_3 = \sum_{i=1}^r d_i, \quad g_4 = \sum_{i=1}^r i^2 \cdot d_i.$$

This gives us the following array, in which blank cells indicate positions not used by this construction, not the value zero.

s_1	s_2	s_3	s_4	1	c_1	d_1
s_5	s_6	s_7	s_8	2	c_2	d_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$s_{\ell-3}$	$s_{\ell-2}$	$s_{\ell-1}$	s_ℓ	r	c_r	d_r
100	200	300	400	-	-	-
e_1	e_2	e_3	e_4	-	g_1	g_3
f_1	f_2	f_3	f_4	-	g_2	g_4

Each corner value admits an equivalent computation via the row code applied to the e or f row, up to a known public constant:

$$g_1 = \left(\sum_{m=1}^4 e_m \right) - 1000 + (1 + \dots + r), \quad g_2 = \left(\sum_{m=1}^4 f_m \right) + (1^3 + \dots + r^3),$$

$$g_3 = \sum_{m=1}^4 m^2 e_m - 10000, \quad g_4 = \sum_{m=1}^4 m^2 f_m,$$

where $1000 = 100+200+300+400$ and $10000 = 1^2 \cdot 100 + 2^2 \cdot 200 + 3^2 \cdot 300 + 4^2 \cdot 400$. Recomputing a corner value both ways and comparing gives the Sharekeeper an

independent, purely additive consistency check, in the same spirit as the original global integrity check, and also catches a $g_1 \leftrightarrow g_3$ or $g_2 \leftrightarrow g_4$ mixup. To simplify the effort, the sums $\sum_{j=1}^r j$ and $\sum_{j=1}^r j^3$ can be provided as a table in r .

The row-share number j additionally binds g_1 to its share: $g'_1 = g_1 + j \pmod{2053}$; only g'_1 , not g_1 , is serialized on the share, so that a mislabeled share fails this check during Recovery Step R1. $\square$

Together, the pairs (c_i, d_i) and (e_m, f_m) enable single-error correction within a row or column (Appendix C), not merely its detection.

Theorem 4 (Minimum Detection Distance). *For any $\ell \in \{12, 18, 24\}$, consider the extended share restricted to its plain checksum layer, $\mathbf{t} = (\mathbf{s}, \mathbf{c}) \in \mathbb{F}_{2053}^N$ of size $N = \ell + r + 5$, consisting of the raw values $\mathbf{s} = (s_1, \dots, s_\ell)$ and the plain checksums $\mathbf{c} = (c_1, \dots, c_r, e_1, \dots, e_4, g_1)$. The check matrix acting on $\mathbf{t}$ has rank $r + 5$, and the minimum weight of a nonzero undetectable error vector is exactly $d_{\min} = 4$, independent of ℓ . That is, all error patterns affecting 1, 2, or 3 cells are detected with certainty.*

Empirical studies of manual data entry, originating with Verhoeff's analysis of the Dutch postal system [13] [5], consistently find that errors cluster into a small number of structured patterns: single-symbol substitution dominates, followed by adjacent transposition ($ab \rightarrow ba$), with twin, jump-transposition, and phonetic errors trailing well behind.

Corollary 1 (Transposition detection). *Any transposition of two distinct grid positions has weight ≤ 2 and is therefore detected with certainty by Theorem 4, independent of any error-distribution assumption.*

We now analyze the full system, including the RS checksums.

Theorem 5 (Minimum Distance, RS Layer). *Consider the full expanded share, consisting of the $r \times 4$ block of raw data cells $\mathbf{s} = (s_1, \dots, s_\ell)$, its row checksums $(c_i, d_i)_{i=1}^r$, column checksums $(e_m, f_m)_{m=1}^4$, and the four corner values g_1, g_2, g_3, g_4 . Since each row is protected by an RS(6, 4) code and each column by an RS($r+2$, r) code, both of minimum distance 3, and since the corner block extends both codes consistently to every row and every column of the resulting grid (Remark [Redundancy]), the whole construction is a genuine two-dimensional product code with minimum distance*

$$d_{\min} = 3 \times 3 = 9,$$

independent of ℓ . That is, every error pattern of weight up to 8, anywhere in the stored artifact, is detected with certainty, and any pattern with at most one error per row and at most one error per column is correctable by independent row-then-column syndrome decoding (Appendix C).

Corollary 2 (Checksums add no information). *For any r , conditional on the $t < k$ raw word-share values held by an unauthorized coalition, the row checksums, column checksums, RS checksums, and corner values are all deterministic public functions of those word shares. Adjoining them to the coalition's view therefore preserves the identical-distribution property of Theorem 3.*

Using authentication codes Each share can be *sealed* with a Message Authentication Code (MAC). This optional step enables the Principal to periodically audit individual shares, without requiring the recovery of the secret or the combination of multiple shares. The per-row MAC covers only the raw data $s_{4j-3}, \dots, s_{4j}$, not the row's own checksums c_j, d_j . This costs nothing in security: checksums are public, deterministic functions of the raw values, so tampering with a checksum alone is already caught by checksum validation (Step S2), and forging the MAC for tampered raw data is exactly as hard whether or not checksums also sit in its domain.

Step S3: Sealing an Expanded Share with a MAC

Input: A raw share $\mathbf{s} = (s_1, \dots, s_\ell)$, arranged into r rows, each of size 4 (again we are dropping the share index, (j) , in the notation).

Output: r message authentication tags $T_1, \dots, T_r \in \mathbb{F}_{2053}$, one per row.

Procedure:

1. Generate $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ uniformly at random, shared across all r rows, and r independent random values $\mu_1, \dots, \mu_r$. Store both on the Manifest; never serialize them on the share.
2. For each row $j = 1, \dots, r$: $T_j = \mu_j + \lambda_1 s_{4j-3} + \lambda_2 s_{4j-2} + \lambda_3 s_{4j-1} + \lambda_4 s_{4j} \pmod{2053}$.
3. Append $T_1, \dots, T_r$ to the share. □

Theorem 6 (Row MAC). *For each row $j = 1, \dots, r$, the tag*

$$T_j = \mu_j + \lambda_1 s_{4j-3} + \lambda_2 s_{4j-2} + \lambda_3 s_{4j-1} + \lambda_4 s_{4j} \pmod{2053}$$

is an affine inner-product one-time MAC in the Carter–Wegman family. Even though the coefficients $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are shared across all r rows, an adversary who does not know $(\boldsymbol{\lambda}, \mu_1, \dots, \mu_r)$ cannot forge a valid tag for a modified row, except with probability at most $1/2053$, regardless of how many other rows' tags have already been observed, and regardless of the adversary's computational resources.

Proof. Fix any candidate $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ and any observed tags $T_1, \dots, T_r$. For every row j , the equation $T_j = \mu_j + \lambda_1 s_{4j-3} + \lambda_2 s_{4j-2} + \lambda_3 s_{4j-1} + \lambda_4 s_{4j}$ determines μ_j uniquely once $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ is fixed. Since μ_j ranges uniformly and independently over $\mathbb{F}_{2053}$ for every row, *every* candidate $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ is equally consistent with the observed tags: the values $\mu_1, \dots, \mu_r$ that would explain them always exist, for any candidate key. Hence observing $T_1, \dots, T_r$ leaves the posterior distribution of $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ identical to its prior, the uniform distribution over $\mathbb{F}_{2053}^4$, and no information about $\boldsymbol{\lambda}$ is leaked by any number of rows. Consequently, an adversary attempting to forge a tag T'_j for a modified row $(s'_{4j-3}, s'_{4j-2}, s'_{4j-1}, s'_{4j}) \neq (s_{4j-3}, s_{4j-2}, s_{4j-1}, s_{4j})$ gains no advantage from having observed other rows' tags: their best strategy is an unaided guess of T'_j (or equivalently of $\boldsymbol{\lambda}$), which succeeds with probability exactly $1/2053$, since T'_j ranges uniformly over $\mathbb{F}_{2053}$ as $\boldsymbol{\lambda}$ ranges uniformly, for any fixed, distinct row content. □

The construction is linear in the row's raw data $s_{4j-3}, \dots, s_{4j}$, which preserves the algebraic structure of the underlying checksums. It does not interfere with the error-detection or error-correction properties of the checksum layer; rather, the two layers are complementary: the **checksum layer** adds redundancy in order to detect and localize accidental transcription and arithmetic errors, while the **MAC layer** detects deliberate tampering with or substitution of a share.

4 The Recovery Ceremony

Step R1: Validating and Auditing the Collected Shares

Input: k authenticated shares $\bar{\mathbf{t}}^{(x_1)}, \dots, \bar{\mathbf{t}}^{(x_k)}$, collected from k distinct Sharekeepers at distinct indices $x_1, \dots, x_k$, together with the Manifest.

Output: k authenticated, consistent expanded shares $\mathbf{t}^{(x_1)}, \dots, \mathbf{t}^{(x_k)}$, ready for interpolation.

Procedure: For each collected share (superscripts dropped for simplicity):

1. **Checksum validation.** Verify $c_1, \dots, c_r, d_1, \dots, d_r, e_1, \dots, e_4, f_1, \dots, f_4, g_1, g_2, g_3, g_4$ against $s_1, \dots, s_\ell$, using Step S2's formulas.
2. **MAC audit.** Retrieve $(\boldsymbol{\lambda}, \mu_1, \dots, \mu_r)$ from the Manifest and recompute $T'_\rho = \mu_\rho + \lambda_1 s_{4\rho-3} + \lambda_2 s_{4\rho-2} + \lambda_3 s_{4\rho-1} + \lambda_4 s_{4\rho} \pmod{2053}$ for each row ρ ; reject the share if $T'_\rho \neq T_\rho$ for any ρ .

□

A share failing the first check must be inspected; maybe it can be corrected. A share subsequently failing the second check is evidence of tampering and should not be trusted.

Theorem 7 (Recovery input integrity). *If all k shares pass both checks of Step R1, then, by Theorem 6, with probability at least $1 - 1/2053$ per row against a computationally unbounded forger, and with certainty against passive corruption of weight at most 8 per share (Theorem 5), the k shares are authentic and internally consistent.*

Reconstructing the Expanded Share by Lagrange Interpolation.

The manual recovery of a Shamir secret can be greatly simplified by precomputing the Lagrange coefficients for the specific set of share indices. For a (k, n) scheme, the secret is recovered as $s = \lambda_1 y_1 + \lambda_2 y_2 + \dots + \lambda_k y_k \pmod{2053}$ where the Lagrange coefficients λ_j are defined as

$$\lambda_j = \prod_{\substack{m=1 \\ m \neq j}}^k \frac{x_m}{x_m - x_j} \pmod{2053}.$$

They are defined by the (known) x -values of the shares, $x_1, \dots, x_k$, and the field size, while they do not depend on the (secret) y -values $y_1, \dots, y_k$. So they can be precomputed for all possible share combinations and included in a small table. Appendix B shows this.

For example, with three shares with x -values 1, 3 and 5 over $\mathbb{F}_{2053}$, the Lagrange coefficients can be computed as $\lambda_1 = 1285, \lambda_3 = 512, \lambda_5 = 257$. If these coefficients are known, the recovery of the secret s now reduces to a single linear combination: $s = 1285 \cdot y_1 + 512 \cdot y_3 + 257 \cdot y_5 \pmod{2053}$. This simplification eliminates the need to perform any modular inversion during recovery.

Step R2: Reconstructing the Expanded Share by Interpolation

Input: k validated expanded shares $\mathbf{t}^{(x_1)}, \dots, \mathbf{t}^{(x_k)} \in \mathbb{F}_{2053}^N$, and the corresponding indices $x_1, \dots, x_k$.

Output: The recovered expanded value $\hat{\mathbf{t}} = (\hat{t}_1, \dots, \hat{t}_N)$.

Procedure:

1. Compute or look up the Lagrange coefficients $\gamma_1, \dots, \gamma_k$ for the recovery set.
2. For each position $i \in \{1, \dots, N\}$: $\hat{t}_i = \sum_{j=1}^k \gamma_j t_i^{(x_j)} \pmod{2053}$. $\square$

By Theorem 3, this recovers $\mathbf{t}$ exactly whenever the k input shares are genuine, since each t_i is a value of a degree- $(k-1)$ polynomial in X .

Step R3: Verifying the Recovered Sequence

Input: The recovered expanded value $\hat{\mathbf{t}} = (\hat{s}_1, \dots, \hat{s}_\ell; \hat{c}_1, \dots, \hat{c}_r, \hat{d}_1, \dots, \hat{d}_r, \hat{e}_1, \dots, \hat{e}_4, \hat{f}_1, \dots, \hat{f}_4, \hat{g}_1, \hat{g}_2, \hat{g}_3, \hat{g}_4)$.

Output: TRUE or FALSE, and, on success, the recovered mnemonic.

Procedure: Parse $\hat{\mathbf{t}}$ into $\hat{\mathbf{s}}$ and its checksum part, and verify $\hat{\mathbf{s}}$ against the recovered checksums using Step S2's formulas. A *pass* confirms the k input shares came from a single, consistent sharing ceremony. So convert $\hat{\mathbf{s}}$ to BIP39 words via the word list [7] to obtain the recovered mnemonic. A *failure* most likely indicates an arithmetic mistake made during Step R2's interpolation, since the k shares already passed Step R1 individually; re-check the interpolation arithmetic first. If the mistake persists, the k input shares may not all have been drawn from a consistent set of Shamir polynomials. Proceed with caution. $\square$

Theorem 8 (Recovery correctness). *Suppose Step R1 succeeds for all k collected shares, and Step R3 succeeds on the resulting recovered sequence. Then, except with probability at most $1/2053$ per row against a computationally unbounded forger, and with certainty against passive corruption of weight at most 8 per share (Theorem 5), the k shares are authentic and were all drawn from a single, consistent sharing ceremony, and the recovered sequence $\hat{\mathbf{s}}$ equals $\mathbf{w}$ exactly. Any such corruption confined to at most one error per row and one error per column is additionally correctable by independent row-then-column syndrome decoding (Appendix C). The forging bound against an unbounded adversary is unaffected by this stronger detection and correction capability, since the checksum layer adds no authentication.*

Remark 1 (A simpler, detection-only variant remains available). Step R1’s checksum validation, as stated, verifies the full checksum inventory $(c, d, e, f, g_1, \dots, g_4)$, giving the full $d_{\min} = 9$ guarantee of Theorem 5. A Sharekeeper who prefers to avoid the RS-style, squared-weight arithmetic may instead verify only the plain checksums $c_1, \dots, c_r, e_1, \dots, e_4, g_1$, skipping d_i, f_m, g_2, g_3, g_4 entirely. By Theorem 4, this simpler check still guarantees detection, with certainty, of any error pattern of weight up to 3 – using nothing beyond plain addition.

Post-recovery validation

Theorem 6 bounds MAC-forgery per modified row, not the overall odds of a bad recovery passing undetected. Two further, computation-dependent checks close some of that gap. First, the Principal enters the recovered mnemonic into air-gapped BIP39-aware software and validates its checksum: for a random incorrect mnemonic, this catches the error with probability 15/16, 63/64, or 255/256 for 12, 18, or 24 words respectively. Second, a truncated, unused wallet address recorded on the share or Manifest at setup time (e.g. `bc1q1234...abcd1234`) can be re-derived from the recovered mnemonic and compared, using compatible wallet software.

5 Conclusion

DuraShare is designed to accommodate a broad range of users, from the security-conscious individual performing every step by hand with paper and a calculator, to the everyday user who relies entirely on software-assisted tooling, to the technically curious who experiment with alternative implementations, to institutional custodians who require additional layers of protection. These profiles are not mutually exclusive: the same underlying arithmetic supports all of them simultaneously.

Observe that the techniques presented in our research are not specific to BIP39 mnemonics. The same Shamir secret sharing, checksum layer, and MAC construction apply equally to secret-sharing a password or passphrase, a safe combination or any other short sequence, though perhaps using a different field. A person may need to protect and later recover such a sequence without relying on any particular piece of technology.

As more and more of our identities and transactions move online, the need for reliable, long-lasting backups only grows. Hardware- and software-based solutions carry their own risks: vendor lock-in, obsolescence, and, ultimately, the very software dependence they were meant to eliminate. Another recent concern is powerful AI. So existing solutions are not always acceptable to every custodian or in every context. Digital sovereignty, the ability to control and recover one’s own digital assets is only becoming more important as this dependence deepens. We believe this paper contributes toward solving this problem.

5.1 Acknowledgment

The second author thanks Jean Martina (LabSEC, INE–UFSC), whose review of an earlier version identified active recovery sabotage and share substitution as the protocol’s principal operational risks. His observation led directly to the inclusion of a MAC as an additional protection.

References

1. Blockchain Commons: bc-sskr security review. GitHub, <https://github.com/BlockchainCommons/bc-sskr/blob/master/SECURITY-REVIEW.md>
2. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. *Journal of Computer and System Sciences* **18**(2), 143–154 (1979)
3. Coinkite: SeedXOR: XOR-based secret sharing for BIP-39 mnemonics. Coldcard documentation (2021), <https://seedxor.com/>
4. Elias, P.: Error-free coding. *IRE Transactions on Information Theory* **4**(4), 29–37 (1954)
5. Gallian, J.A.: Error detection methods. *ACM Computing Surveys* **28**(3), 504–517 (1996)
6. McNally, W., Allen, C.: UR type definition for sharded secret key reconstruction (SSKR). Blockchain Commons Research, BCR-2020-011, v1.0.1 (2020)
7. Palatinus, M., Rusnak, P., Voisine, A., Bowe, S.: BIP-39 english wordlist. Bitcoin Improvement Proposals, GitHub (2013), <https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt>
8. Palatinus, M., Rusnak, P., Voisine, A., Bowe, S.: BIP-39: Mnemonic code for generating deterministic keys. Bitcoin Improvement Proposals (2013), <https://bips.dev/39/>
9. Poelstra, A., O’Connor, L., Corallo, S.: BIP-93: codex32: Checksummed SSSS-aware BIP32 seeds. Bitcoin Improvement Proposals, Draft (2023), <https://bips.dev/93/>
10. Rusnak, P., Kozlik, A., Vejpustek, O., Susanka, T., Palatinus, M., Hoenicke, J.: SLIP-0039: Shamir’s secret-sharing for mnemonic codes. SatoshiLabs Improvement Proposals, Final (2019), <https://github.com/satoshilabs/slips/blob/master/slip-0039.md>
11. Shamir, A.: How to share a secret. *Communications of the ACM* **22**(11), 612–613 (1979)
12. Snead, P.: The codex32 mathematical companion. secretcodex32.com (2023), <https://secretcodex32.com/docs/2023-08-23-math.pdf>
13. Verhoeff, J.: Error detecting decimal codes. Tech. rep., Mathematisch Centrum, Amsterdam (1969)
14. Wegman, M.N., Carter, J.L.: New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences* **22**(3), 265–279 (1981)

A Card-Based Randomness: Full Procedure

This appendix gives the mechanics behind the rejection-sampling procedure summarized in Section 2.5.

Note that 2052 corresponds to 6660 , so $2053 \leftrightarrow 6661, \dots, 2057 \leftrightarrow 6665$. This representation has the advantage that integers larger than 2053, which need to be rejected, are easy to recognize.

Using cards, we assign the four suits as follows: $(\clubsuit, \heartsuit, \spadesuit, \diamondsuit) = (d_3, d_2, d_1, d_0)$. A King represents 0 , whereas Ace through 6 represent 1 through 6 . The card $\diamondsuit 6$ is excluded, so the resulting deck contains 27 cards. A printed reference sheet may show the value assigned to each card, so the calculation requires only addition.

Before any calculations, the Principal applies a visual rejection rule. Three 6s and a King represent 2052, so the draw is valid, but three 6s and any other card represent an overflow. Such a draw is rejected and a fresh attempt begins. Any other draw is valid.

If valid, the four weighted values are added and recorded. All cards are then returned and the reduced deck reshuffled before the next attempt.

For each attempt, the Principal thoroughly shuffles the reduced deck and places it face down. Thorough shuffling is essential; we recommend washing the cards face down on a clean surface for at least one minute, as casinos do. Cards are revealed from the top until all four suits are represented, retaining only the first card of each suit.

B Lagrange coefficients for common parameters

Scheme	Shares	Coeff.	Scheme	Shares	Coeff.
2-of-3	(λ_1, λ_2)	$(2, 2052)$	3-of-5	$(\lambda_1, \lambda_2, \lambda_3)$	$(3, 2050, 1)$
	(λ_1, λ_3)	$(1028, 1026)$		$(\lambda_1, \lambda_2, \lambda_4)$	$(687, 2051, 1369)$
	(λ_2, λ_3)	$(3, 2051)$		$(\lambda_1, \lambda_2, \lambda_5)$	$(1029, 1367, 1711)$
2-of-4	(λ_1, λ_2)	$(2, 2052)$		$(\lambda_1, \lambda_3, \lambda_4)$	$(2, 2051, 1)$
	(λ_1, λ_3)	$(1028, 1026)$		$(\lambda_1, \lambda_3, \lambda_5)$	$(1285, 512, 257)$
	(λ_1, λ_4)	$(1370, 684)$		$(\lambda_1, \lambda_4, \lambda_5)$	$(686, 1367, 1)$
	(λ_2, λ_3)	$(3, 2051)$		$(\lambda_2, \lambda_3, \lambda_4)$	$(6, 2045, 3)$
	(λ_2, λ_4)	$(2, 2052)$		$(\lambda_2, \lambda_3, \lambda_5)$	$(5, 2048, 1)$
	(λ_3, λ_4)	$(4, 2050)$		$(\lambda_2, \lambda_4, \lambda_5)$	$(1372, 2048, 687)$
				$(\lambda_3, \lambda_4, \lambda_5)$	$(10, 2038, 6)$

Table 1. Lagrange coefficients in $\mathbb{F}_{2053}$. Note their sum always equals 1 modulo 2053.

C Manually Correcting Reed–Solomon Syndromes

This appendix illustrates the RS(8,6) syndrome-based error correction described in Section 2.7 using concrete numbers. The same principle applies to RS(7,5), RS(6,4), and RS(5,3), with correspondingly shorter equations. Computations are performed modulo $p = 2053$, but nothing here depends on that specific choice: the same formulas work over any prime field.

A row of 6 indices $m_1, \dots, m_6$ is protected by two check symbols symbols,

$$p_1 = \sum_{i=1}^6 i \cdot m_i \pmod{2053}, \quad p_2 = \sum_{i=1}^6 i^2 \cdot m_i \pmod{2053}.$$

Considering the six indices $\mathbf{m} = (214, 897, 1543, 60, 1200, 333)$ we can computing the check symbols as $(504, 1100)$.

Suppose the Sharekeeper mistranscribes the fourth word, writing 837 instead of 60 – an error of magnitude 777 at position 4. So the received row is $(214, 897, 1543, \mathbf{837}, 1200, 333, 504, 1100)$. Recomputing the syndromes from this received row:

$$\begin{aligned} S_1 &= (1 \cdot 214 + 2 \cdot 897 + 3 \cdot 1543 + 4 \cdot 837 + 5 \cdot 1200 + 6 \cdot 333) - 504 \\ &\equiv 1055 \pmod{2053}, \end{aligned}$$

$$\begin{aligned} S_2 &= (1^2 \cdot 214 + 2^2 \cdot 897 + 3^2 \cdot 1543 + 4^2 \cdot 837 + 5^2 \cdot 1200 + 6^2 \cdot 333) - 1100 \\ &\equiv 114 \pmod{2053}. \end{aligned}$$

Since both S_1 and S_2 are nonzero, the error lies in a data position. To find it, add S_1 to itself repeatedly, comparing against S_2 after each step. Adding $S_1 = 1055$ to itself repeatedly gives 1055, 57, 1112, 114. The match after 4 steps identifies the error position $i = 4$. The magnitude follows from $e = S_1/i = 1055/4 = 1055 \cdot 1540 \equiv 777 \pmod{2053}$, a single multiplication by 4^{-1} , taken from a small table, thus recovering exactly the injected error. The Sharekeeper corrects position 4 by subtracting 777, restoring the original value 60. All of this is arithmetic well within reach of a calculator.

i	1	2	3	4	5	6
$i^{-1} \pmod{2053}$	1	1027	1369	1540	1232	1711

Observe that since the syndrome (S_1, S_2) depends only on the error pattern and not on the codeword itself, this computation could safely be delegated to an untrusted or online device: even if fully exposed, the syndrome reveals nothing about the actual data values being protected. One could even extend this technique to correct 2 errors, using an $(n+4, n)$ RS code with four check symbols, for which manual calculation is not an option.

D Suggested improvements on SeedXOR

In SeedXOR, each 11-bit index is divided in three nibbles of size 3, 4 and 4 which are transcribed to hex. by consulting the word list. The computation of the

bitwise XOR is then performed for each of the three hex symbols by consulting a 16×16 table. SeedXOR has no protection whatsoever against errors during the computations, while its complicated arithmetic increases the probability of mistakes considerably, with potentially catastrophic effects.

One simple way to improve SeedXOR would be to implement the one-time pad not modulo 2, but modulo 2048. The last word from the list, *zoo*, would be mapped to 0, and addition mod 2048 can be performed by using a basic pocket calculator. In addition, we can strengthen the computations by using checksums, as presented in Section 2.6, which are purely additive.

Alternatively one could implement the one-time pad using addition modulo 2053. To detect errors we use a RS(8,6) code grouping 6 (say) indices together. Since RS codes are linear, the syndrome symbols on the cipher text must equal the sum of the syndrome symbols of the plaintext and the key.

m_1	m_2	m_3	m_4	m_5	m_6	−	m_7	m_8	
k_1	k_2	k_3	k_4	k_5	k_6	−	k_7	k_8	
+	+	+	+	+	+	−	+	+	mod 2053
c_1	c_2	c_3	c_4	c_5	c_6	−	c_7	c_8	