White Paper

Q-Reduct

Quantum-Ready Erasure Codec for Extreme Storage Reduction

David N. Halenta

Picyboo Cybernetics Inc., Research Lab (Canada)

Subject: Quantum Computing, Information Theory, Data Compression, Cryptography, Quantum Algorithms, Storage Optimization, Computational Complexity

Contact: desk@davidhalenta.de / contact@picyboo.com

CSR: https://CognitiveScienceResearch.com/CSR4812

ORCID: https://orcid.org/0009-0003-3994-7912

Version 1.2, October 27, 2025.

DOI: https://doi.org/10.5281/zenodo.17360291

GitHub: https://github.com/Picyboo-Cybernetics/picyboo-public-qreduct

CSR-WP02-081025

Abstract

Q-Reduct is a lossless, quantum-ready erasure codec that achieves file-size reduction by deliberately discarding payload bits and retaining a compact set of deterministic constraints that uniquely characterize the original. Decoding is posed as an oracle-search problem: given the stored constraints, find the erased bits that make every local and global check true. Classically, aggressive erasure makes decoding computationally infeasible; on a future quantum computer, amplitude-amplification (Grover-style) reduces the expected search cost from O(2^m) to O(2^(m/2)) oracle calls, where m is the effective constraint strength.

Keywords: quantum decoding, Grover search, oracular reconstruction, erasure codec, Merkle verification, linear syndromes, storage reduction

Projected benefit. Under quantum decoding and suitable parameterization, the size reduction approaches the erased fraction $e = d/k$ per chunk. With overhead amortized, reductions on the order of 80-90% (i.e., encoded size ≤ 10-20% of original) are mathematically achievable for large e (e.g., $e \approx 0.9$) provided the quantum oracle budget suffices to search the induced space and the constraint set enforces uniqueness. Classically, the same parameters are intentionally impractical; small, testable settings validate correctness but do not exceed top classical compressors on average data.

Contributions. We specify the format, constraints, oracle, and verification hierarchy; derive classical vs quantum complexity; give parameter guidance; provide reference pseudocode and a PHP skeleton; and outline a rigorous evaluation plan and a quantum implementation roadmap.

1. Motivation and Scope

Modern compressors are close to Shannon-style limits on many corpora. Surpassing them consistently and losslessly requires adding information (side-info/models) or trading compute for storage. Q-Reduct formalizes the latter: remove bits now, prove uniqueness by constraints, and shift the heavy work to the decoder. The approach is:

- **Lossless by specification.** If decoding succeeds, the output equals the original bit-for-bit.
- **Asymmetric.** Encoding is linear-time; decoding cost scales with the constraint budget, not with encoder effort.
- **Quantum-ready.** The decoding predicate is an explicit oracle suitable for amplitude amplification.

Use cases. Long-horizon archives expecting quantum resources; controlled pipelines willing to accept expensive, delayed decode; research platforms evaluating quantum advantage on real data paths.

Non-goals. Q-Reduct is not a universal, faster-than-zstd drop-in today; it is a storage primitive whose practical advantage manifests only with quantum search (or vast classical compute) at aggressive settings.

2. Model and Notation

- **File size:** N bytes
- **Chunk size:** k bytes (fixed; last chunk may be shorter)
- **Erasure per chunk:** d bytes removed; prefix kept: (k – d) bytes
- **Erasure fraction:** e = d/k
- **Chunks:** C = ⌈N/k⌉
- **Candidate space per chunk:** |S| = 2^(8d)
- **Constraints per chunk:**
  - Digest m bits: dig_i = trunc_m(H(seed ‖ $chunk_i$))
  - Linear syndrome r bits: s_i = H_lin · v_i over GF(2)
  - Chain check t bits across neighbors
- **Global checks:** Merkle root over chunk digests; SHA-256 of the full original
- **Seed:** derived from a stable snapshot (e.g., filename hash, original size, original mtime) and/or a user passphrase

3. High-Level Architecture

Encoding.

Split the file into chunks of size k. For chunk i,

store the prefix $P_i \in \{0,1\}^{(8(k-d))}$.

Compute and store constraints:

m-bit digest, r-bit linear syndrome, optional t-bit chain check to P_{i+1}.

Store a Merkle root over per-chunk digests and a global SHA-256 of the original.

Decoding.

For each chunk i, search for $tail_i \in \{0,1\}^{(8d)}$ such that the oracle O_i (Section 6) returns true. Compose the recovered chunk $C_i = P_i \parallel tail_i$. Verify the global SHA-256. If all chunks succeed and the final hash matches, output is guaranteed equal to the original.

Design principles:

- **Locality.** Oracles are chunk-local to enable parallel search.
- **Composability.** Chain checks and a Merkle hierarchy prune cross-chunk combinations early.
- **Determinism.** No probabilistic reconstruction: constraints define a single valid original with overwhelming probability.

4. Format Specification

4.1. Header

- **magic** (4 B): "QRED"
- **version** (1 B)
- **params** (7 B total): k (uint16), d (uint8), m (uint8), r (uint8), t (uint8), flags (uint8)
- **orig_size** (8 B)
- **global_sha256** (32 B)
- **seed_snapshot_len** (1 B), **seed_snapshot** (≤64 B): compact snapshot and/or passphrase tag
- Optional **hmac** (32 B) if passphrase binding is enabled

Flags. bit0: passphrase used; bit1: chain enabled; bit2: syndrome enabled; bit3: merkle enabled.

4.2. Chunk record i

- **prefix_i:** $(k - d)$ bytes
- **digest_i:** $\lceil m/8 \rceil$ bytes
- **syndrome_i:** $\lceil r/8 \rceil$ bytes (optional)
- **chain_i:** $\lceil t/8 \rceil$ bytes (optional)

4.3. Trailer

- **merkle_root** (if enabled)
- Optional auxiliary indices

5. Constraint Primitives

Truncated digest m

- **Role:** probabilistic preimage check
- **Expected matches (per chunk):** E_m = 2^(8d − m)
- **Cost:** ⌈m/8⌉ bytes

Linear syndrome r

- **Role:** algebraic pruning via s_i = H_lin · v_i, H_lin ∈ {0,1}^(r × 8k)
- Reduces the candidate set by a factor of 2^r in expectation (keeps approximately 1/2^r)
- Cheap to compute, easy to compose with digest
- **Cost:** ⌈r/8⌉ bytes

Chain check t

- **Role:** link neighbors to prevent cross-chunk combinatorics
- Example: trunc_t(H(P_i ‖ P_{i+1}))
- **Cost:** ⌈t/8⌉ bytes

Merkle hierarchy

- **Role:** hierarchical pruning and global integrity
- **Overhead:** compact (root only) if per-chunk digests are stored anyway

Per-chunk net saving (bytes):

$\Delta\_chunk \approx d - (m + r + t)/8$

Whole file net:

$\Delta\_total \approx C \cdot \Delta\_chunk$ – header/trailer overhead

Amortize per-chunk control bits by increasing k for the same fractional erasure e.

6. Decoding as an Oracle Problem

For chunk i, define the oracle O_i that tests a candidate tail:

Input: P_i (stored), candidate X ∈ {0,1}^(8d), neighbor P_{i+1} if chain is enabled, seed.

Construct: $C_i = P_i \parallel X$

Return true iff:

1. trunc_m(H(seed ‖ C_i)) = digest_i
2. If enabled, H_lin · C_i = s_i
3. If enabled, trunc_t(H(P_i ‖ P_{i+1})) = chain_i
4. (Optional during search) partial Merkle consistency holds

Uniqueness condition

With independent constraints, the expected number of matches is

E[matches] ≈ 2^(8d − m − r_eff − t_eff) ≤ 1

where r_eff and t_eff reflect the effective bits of pruning contributed by the syndrome and chain checks (bounded above by r and t). Choose parameters so E ≤ 1 per chunk.

7. Complexity: Classical vs Quantum

Let $m_{eff} = m + r_{eff} + t_{eff}$.

Classical (random-oracle model): expected oracle calls to find a preimage that passes all checks scale as

$$T_{classical} = \Theta(2^{(m_{eff})})$$

Even if $E \approx 1$, finding the solution typically costs exponential time in m_{eff}.

Quantum (Grover / amplitude amplification):

$$T_{quantum} = \Theta(2^{(m_{eff}/2)})$$

given a coherent oracle U_O implementing the joint predicate. This is the core speedup that makes large d feasible.

Implication

To erase a fraction $e = d/k$ and keep uniqueness with negligible false positives, pick $m_{eff} \approx 8d$. Then

- $T_{classical} \approx 2^{(8d)}$
- $T_{quantum} \approx 2^{(4d)}$

For d = 12 bytes, $T_{classical} \sim 2^{96}$ (impossible), while $T_{quantum} \sim 2^{48}$ oracle calls (still enormous, but a roadmap target for future QC at scale). In practice, choose d to match the oracle budget.

8. Projected Storage Reduction

Let $e = d/k$. Per chunk:

$$\text{size\_stored} = (k - d) + (m + r + t)/8$$

Fractional size:

$$\rho = \text{size\_stored}/k = 1 - e + (m + r + t)/(8k)$$

For fixed (m, r, t) and growing k, overhead amortizes. Under quantum decoding, we select large d (large e) and modest (m, r, t) that still enforce uniqueness, yielding:

$$\rho \approx 1 - e \Longrightarrow \text{reduction} \approx e$$

Example (headline)

Choose $e = 0.90$ (erase 90% of each chunk), $k = 1$ MiB, $(m, r, t) = (64, 16, 16)$ bits.

Overhead $(m + r + t)/8 = 12$ B per chunk $\Longrightarrow \rho \approx 0.10 + 12/2^{20} \approx 0.10001$.
The encoded file is ~10% of the original, i.e., ~90% reduction,
contingent on a quantum decoder capable of handling $m_{eff} \approx 96$ bits of oracle strength per chunk (and thus $T_{quantum} \sim 2^{48}$ coherent oracle evaluations).
Classically infeasible by design.

Today's regime. With small d chosen so classical brute force terminates, ρ is near 1 once (m, r, t) are counted; practical reductions beyond top compressors are not expected on average data.

9. Parameter Guidance

- **Chunk size k.** Larger k better amortizes overhead; increases working-set and I/O. Typical quantum-target: $k \in$ [256 KiB, 4 MiB].
- **Erasure d.** Drives savings. Classical tests: $d \in \{2, 3, 4\}$ bytes. Quantum-target: d into tens or hundreds of bytes (or more), bounded by search budget.
- **Digest m.** Dominant determinant of search cost. For uniqueness, start from $m \approx 8d - \delta$ and add syndrome/chain to close the gap.
- **Syndrome r.** Cheap pruning. Sparse random or LDPC-like H_lin works well; typical $r \in \{8, 16, 32\}$.
- **Chain t.** Helps kill cross-chunk ambiguity; $t \in \{0, 8, 16\}$ suffices.
- **Seeds.** Prefer a stored snapshot (24-64 B). Optional passphrase can be added; neither is used for secrecy unless HMAC is enabled.

Rule of thumb. Positive net saving per chunk requires $d > (m + r + t)/8$ or amortization of control bits across larger k.

10. Security, Robustness, and Portability

- **Integrity.** Global SHA-256 and Merkle ensure deterministic acceptance or failure; no silent corruption.
- **Tamper binding (optional).** HMAC over header with passphrase.
- **Portability.** Include a compact seed snapshot; do not rely on live file system metadata.
- **Adversarial inputs.** Parameterize m_eff to keep $E \leq 1$ with wide safety margins; randomize H_lin per file if desired.

11. Pseudocode (Reference)

11.1 Encoder

```
function QRED_Encode(bytes data, uint16 k, uint8 d, uint8 m, uint8 r, uint8 t, Seed seed):
    N = len(data)
    C = ceil(N / k)
    header = build_header(k,d,m,r,t,N, sha256(data), seed.snapshot)
    out.write(header)

    digests = []
    prefixes = []
    syndromes = []
    chains = []

    for i in 0..C-1:
        chunk = data[i*k : min((i+1)*k, N)]
        if len(chunk) < d: error("chunk shorter than d")
        prefix = chunk[0 : len(chunk)-d]
        full   = chunk

        dig_i = trunc_m( sha256(seed || full), m )
        syn_i = trunc_r( linear_syndrome(H_lin, full), r )
        prefixes.append(prefix); digests.append(dig_i); syndromes.append(syn_i)

    for i in 0..C-1:
        next_prefix = (i+1<C) ? prefixes[i+1] : EMPTY
        chain_i = (t>0) ? trunc_t( sha256(prefixes[i] || next_prefix), t ) : EMPTY
        chains.append(chain_i)

    for i in 0..C-1:
        out.write(prefixes[i])
        out.write(digests[i])
        if r>0: out.write(syndromes[i])
        if t>0: out.write(chains[i])

    if merkle_enabled:
        root = merkle_root(digests)
        out.write(root)
```

11.2 Oracle and Decoder (classical brute, per chunk)

```
function Oracle_Test(prefix, candidate_tail, digest, syndrome, chain, next_prefix, seed, m,r,t):
    full = prefix || candidate_tail
    if trunc_m(sha256(seed || full), m) != digest: return FALSE
    if r>0 and trunc_r(linear_syndrome(H_lin, full), r) != syndrome: return FALSE
    if t>0:
        if trunc_t(sha256(prefix || next_prefix), t) != chain: return FALSE
    return TRUE

function QRED_Decode(stream in):
    header = parse_header(in)
    k,d,m,r,t = header.params
    out = empty_buffer()
    chunks = []
    for i in 0..C-1:
        prefix = in.read(k-d for full chunks; last chunk sized)
        digest = in.read(ceil(m/8))
        syndrome = (r>0) ? in.read(ceil(r/8)) : EMPTY
        chain    = (t>0) ? in.read(ceil(t/8)) : EMPTY
        next_prefix = peek_next_prefix(in)  // or deferred check
        found = FALSE
        for x in 0 .. 2^(8*d)-1:
            tail = int_to_bytes(x, d)
            if Oracle_Test(prefix, tail, digest, syndrome, chain, next_prefix, header.seed, m,r,t):
                chunks.append(prefix || tail)
                found = TRUE
                break
        if not found: error("no candidate for chunk " + i)
    data = concat(chunks)
    if sha256(data) != header.global_sha256: error("global hash mismatch")
    return data
```

12. Implementation Appendix (PHP Skeleton)

Purpose: reference only; not optimized; no quantum paths; suitable for small-d classical validation.

```
<?php
function sha256bin(string $b): string { return hash('sha256', $b, true); }

function trunc_bits(string $bin, int $bits): string {
    $bytes = intdiv($bits + 7, 8);
    $t = substr($bin, 0, $bytes);
    $excess = $bytes * 8 - $bits;
    if ($excess > 0) {
        $last = ord($t[$bytes - 1]) >> $excess;
        $t[$bytes - 1] = chr($last);
    }
    return $t;
}

function lin_syndrome(string $chunk, int $rBits): string {
    // demo: XOR-of-bytes folded into 32-bit, then truncate
    $x = 0;
    $n = strlen($chunk);
    for ($i = 0; $i < $n; $i++) $x ^= ord($chunk[$i]);
    $raw = pack('N', $x);
    $len = intdiv($rBits + 7, 8);
    return substr($raw, 0, $len);
}

function qred_encode(string $inPath, string $outPath, int $k=4096, int $d=3,
                     int $m=24, int $r=8, int $t=8, string $seed=""): void {
    $data = file_get_contents($inPath);
    $N = strlen($data);
    $fh = fopen($outPath, 'wb');
    fwrite($fh, "QRED"); fwrite($fh, chr(1));
    // params
    fwrite($fh, pack('n', $k)); fwrite($fh, chr($d)); fwrite($fh, chr($m));
    fwrite($fh, chr($r)); fwrite($fh, chr($t)); fwrite($fh, chr(0));
    fwrite($fh, pack('J', $N)); fwrite($fh, sha256bin($data));
```

```
    $snap = ""; fwrite($fh, chr(strlen($snap))); // seed snapshot len
    if ($snap !== "") fwrite($fh, $snap);

    $mBytes = intdiv($m + 7, 8); $rBytes = intdiv($r + 7, 8);
    $tBytes = intdiv($t + 7, 8);
    $prefixes = $digests = $syndromes = $chains = [];
    $C = intdiv($N + $k - 1, $k);

    for ($i=0,$off=0; $i<$C; $i++,$off+=$k) {
        $chunk = substr($data, $off, min($k, $N - $off));
        if (strlen($chunk) <= $d) throw new RuntimeException("chunk shorter than d");
        $prefix = substr($chunk, 0, strlen($chunk) - $d);
        $digest = trunc_bits(sha256bin($seed . $chunk), $m);
        $syn = ($r>0) ? trunc_bits(lin_syndrome($chunk, $r), $r) : "";
        $prefixes[] = $prefix; $digests[] = $digest; $syndromes[] = $syn;
    }
    for ($i=0; $i<$C; $i++) {
        $next_prefix = ($i+1<$C) ? $prefixes[$i+1] : "";
        $chain = ($t>0) ? trunc_bits(sha256bin($prefixes[$i] . $next_prefix), $t) : "";
        $chains[] = $chain;
    }
    for ($i=0; $i<$C; $i++) {
        fwrite($fh, $prefixes[$i]);
        fwrite($fh, $digests[$i]);
        if ($r>0) fwrite($fh, $syndromes[$i]);
        if ($t>0) fwrite($fh, $chains[$i]);
    }
    fclose($fh);
}

function qred_decode(string $inPath, string $outPath): void {
    $f = fopen($inPath, 'rb');
    if (fread($f,4)!=="QRED") throw new RuntimeException("bad magic");
    $ver = ord(fread($f,1));
    $k = unpack('n', fread($f,2))[1]; $d=ord(fread($f,1)); $m=ord(fread($f,1));
    $r=ord(fread($f,1)); $t=ord(fread($f,1)); $flags=ord(fread($f,1));
    $N = unpack('J', fread($f,8))[1]; $sha = fread($f,32);
    $snapLen = ord(fread($f,1)); $seed = ($snapLen>0) ? fread($f,$snapLen) : "";
    $mBytes = intdiv($m+7,8); $rBytes = intdiv($r+7,8); $tBytes = intdiv($t+7,8);
```

```
    $C = intdiv($N + $k - 1, $k);
    $out = "";
    for ($i=0; $i<$C; $i++) {
        $lastSize = ($i<$C-1) ? $k : ($N - $k*($C-1));
        $prefixLen = $lastSize - $d;
        $prefix = fread($f, $prefixLen);
        $digest = fread($f, $mBytes);
        $syn = ($r>0)? fread($f, $rBytes):"";
        $chain = ($t>0)? fread($f, $tBytes):"";
        // Peek next prefix for chain check
        $pos = ftell($f);
        $next_prefix = "";
        if ($i<$C-1) {
            $nextSize = ($i+1<$C-1) ? $k : ($N - $k*($C-1));
            $next_prefix = fread($f, $nextSize - $d);
        }
        fseek($f, $pos);
        $found = null;
        $space = 1 << (8*$d);
        for ($x=0; $x<$space; $x++) {
            $tail = "";
            for ($b=$d-1; $b>=0; $b--) $tail = chr(($x >> (8*$b)) & 0xFF) . $tail;
            $full = $prefix . $tail;
            if (trunc_bits(sha256bin($seed.$full), $m) !== $digest) continue;
            if ($r>0 && trunc_bits(lin_syndrome($full,$r), $r) !== $syn) continue;
            if ($t>0) {
                if (trunc_bits(sha256bin($prefix.$next_prefix), $t) !== $chain) continue;
            }
            $found = $full; break;
        }
        if ($found===null) throw new RuntimeException("no candidate in chunk $i");
        $out .= $found;
    }
    fclose($f);
    if (sha256bin($out) !== $sha) throw new RuntimeException("global hash mismatch");
    file_put_contents($outPath, $out);
}
```

Notes. This skeleton keeps the critical ideas compact. Real implementations should stream I/O, validate bounds, and add Merkle verification and error handling.

13. Evaluation Plan

Datasets. Text/JSON/logs (structured), uncompressed binaries, already-compressed (control), random (control).

Baselines. zstd-19, brotli-11, paq/paq8.

Parameters. $k \in \{1 \text{ KiB}, 4 \text{ KiB}, 64 \text{ KiB}\}$, $d \in \{2, 3, 4, 6\}$, $m \approx 8d$, $r \in \{8, 16\}$, $t \in \{0, 8\}$.

Metrics. Encoded size ratio ρ; classical decode attempts per chunk;
wall-clock decode time; false-match rate; pass/fail on hash.

Success criteria (classical lab). Deterministic correctness on small d;
empirical match to predicted costs; no silent failures.

Quantum roadmap validation. Map the oracle to a reversible circuit; estimate gate counts for SHA-256 truncation and linear checks; extrapolate feasible (d, m_{eff}) with assumed quantum resources.

14. Quantum Implementation Roadmap

Oracle unitary U_O. Build a coherent predicate for (digest, syndrome, chain).

- SHA-256 reversible circuits are known in literature; use truncation by discarding high bits.
- Linear syndromes are Clifford-friendly.

Amplitude amplification. Standard Grover iterations; per-chunk search in superposition space $|x\rangle$, $x \in \{0,1\}^{(8d)}$.

Data access. Seed and stored constants are classical; embed as control parameters. Prefix and neighbor prefixes are classical inputs loaded to quantum ancillas.

Parallelization. Run per-chunk searches in parallel across quantum processors; or pipeline chunks.

Resource estimation. For target d and m_eff, estimate $2^{(m\_eff/2)}$ oracle calls; compute depth × calls; incorporate error-correction overheads.

Hybrid pruning. Classical prefilters (e.g., linear checks) can be pushed ahead to lower oracle load.

15. Limitations and Risks

- **Impossibility bounds.** This is not a universal better-than-entropy compressor; it trades compute for storage and relies on constraints, not magic.
- **Classical practicality.** Aggressive settings are intentionally infeasible today.
- **Quantum uncertainty.** Timelines and practical oracle costs are uncertain; resource estimates must be revisited.
- **Portability.** If seed snapshots or passphrases are lost, decoding fails by design.
- **Adversarial inputs.** Choose conservative m_eff and randomized matrices to bound worst-case collisions.

16. Conclusion

Q-Reduct reframes lossless storage as compute-for-storage exchange with an explicit quantum decoding advantage. By erasing a large fraction of payload and storing compact constraints, we shift complexity to a search oracle. Classically, only small d is testable; under quantum decoding, projected reductions track the erasure fraction, enabling on the order of 80-90% size reduction for high e with sufficient quantum resources and properly tuned constraints. The specification, pseudocode, PHP skeleton, and evaluation plan provided here support immediate laboratory validation and future quantum mapping.

Acknowledgments

The author thanks the cognitive neuroscience and artificial intelligence research communities for decades of foundational work that makes conceptual synthesis possible. Special acknowledgment to the pioneering work on complementary learning systems, meta-cognition, and conflict monitoring that inspired this architectural framework.

Disclosure Statements

Conflict of Interest Statement: The author declares no conflicts of interest.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study.

Context of Publication

This document was prepared by Picyboo Cybernetics Inc. (CA) as part of its ongoing research and development activities in foundational computational architectures integrating neural-cognitive mechanisms for next-generation systems. It reflects theoretical and technical considerations derived from internal experimentation and design work related to Picyboo technologies and affiliated products. While not exhaustive, it represents a portion of the company's broader research efforts and is shared publicly to contribute to open scientific and technological discourse.

License and Authorship Notice

Appendix A — Symbols at a Glance

Symbol	Definition
N	Original size (bytes)
k	Chunk size (bytes)
d	Bytes erased per chunk
e	Erasure fraction = d/k
m, r, t	Digest, syndrome, chain bits
C	Chunk count = ⌈N/k⌉
Δ_chunk	Net saving per chunk = d − (m+r+t)/8
E	Expected matches ≈ 2^(8d − m − r_eff − t_eff)
m_eff	Effective constraint strength = m + r_eff + t_eff
T_classical	Classical complexity = Θ(2^(m_eff))
T_quantum	Quantum complexity = Θ(2^(m_eff/2))
ρ	Fractional size = size_stored/k

Appendix B — Parameter Tables (examples)

Per-chunk net saving (bytes): Δ_chunk = d – (m+r+t)/8

Note on notation: In mathematical formulas we use ‖ for concatenation; in pseudocode and implementation we use ||.

Classical-testable parameters (small d)

k (Bytes)	k (KiB)	d (Bytes)	e = d/k	m	r	t	Δ_chunk (B)	ρ	Reduction
4,096	4	3	0.000732	24	8	8	-2	≈ 1.00049	-0.049%
65,536	64	6	0.000092	48	16	16	-4	≈ 1.00006	-0.006%

Quantum-target regime (90% headline example)

k (Bytes)	k (KiB)	d (Bytes)	e = d/k	m	r	t	Δ_chunk (B)	ρ	Reduction
1,048,576	1	943,718	0.90	64	16	16	943,706	≈ 0.10001	≈ 90.0%

In the quantum regime: e ≈ 0.9, overhead (m+r+t)/8 ≪ k, enabling dramatic reduction with quantum oracle budget.

Appendix C — Pseudocode: Merkle and Syndrome Construction

```
function merkle_root(list digests):
  level = digests
  while len(level) > 1:
    next = []
    for j in 0..step2(len(level)):
      a = level[j]
      b = (j+1<len(level)) ? level[j+1] : level[j]
      next.append( sha256(a || b) )
    level = next
  return level[0]

function random_sparse_Hlin(r, bitsPerChunk, density):
  // return r x bitsPerChunk binary matrix with given sparsity
```

Appendix D — Implementation Checklist

- Streaming encoder/decoder; bounds checks; endian consistency
- Stable seed snapshot; optional passphrase; deterministic seed derivation
- Robust header validation; versioning
- Optional Merkle root; chunk digests already present
- Unit tests: oracle truth table; collision statistics; chunk independence
- Benchmarks: per-chunk oracle cost; end-to-end decode time; corpus ratios
- Security: HMAC option; tamper detection; controlled failure semantics