CQRC: Differential Testing Two Zero-Dependency QR Code Encoders at the Module-Matrix Level

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com
https://github.com/yxpil/CQRC

Abstract

QR Code encoders are usually validated by scanning a rendered image, a test that lets the decoder's error correction absorb encoder mistakes and that samples few of the possible inputs. This paper reports CQRC, a QR Code generator written twice from scratch—in C# and in JavaScript—without third-party dependencies, and evaluates it by taking the module matrix, not the image, as the conformance unit. On a seeded corpus of 200 inputs spanning versions 1–40, all four error-correction levels and five payload families, the two implementations produce bit-identical matrices for 192 inputs (1,164,776 modules); the other eight expose a C# defect. A third-party oracle (python-qrcode 8.2, invoked with an explicit mask) agrees on 60/60 cases (585,988 modules), a 40×4 structural sweep passes in both languages, and byte-mode capacity matches the published ISO table at all 44 checked points. We diagnose the eight failures exactly: the C# Reed–Solomon encoder indexes a polynomial whose leading zero coefficients have been stripped, so any block whose remainder starts with `0x00` throws (8/200 cases; the predictor has accuracy 1.000, against 8.95 expected occurrences in 2,292 blocks). The JavaScript port supplies the missing zero padding and is correct. A second defect, a PNG IDAT stream without an Adler-32 checksum, is rejected by both CPython and Node zlib despite valid chunk CRCs. We also measure the metadata trailer (20/20 recovered from the code words; its "ISO/IEC 21570" label is a misattribution, since ISO 21570 is a food-analysis standard) and the geometric cost of styling: circular finders change 16 of 49 cells per finder, and 0.5 module rounding removes 40% of a module's area. No decoder was available, so we do not claim that styled symbols scan.

Artifact. The two implementations, the golden vectors, the measurement harness and every result file are open source at `https://github.com/yxpil/CQRC`. The repository is treated as read-only input: the defects in Section 4.5 are reported, not patched. All numbers below come from `scripts/results/*.json`, produced by `measure_cqrc.py` on the toolchain recorded in Section 7; no package was installed from the network.

1 Introduction

QR Codes are the most widely deployed two-dimensional barcode, and a QR encoder is small enough to be reimplemented from the specification in a few thousand lines. CQRC is one such project: a C# library and a JavaScript port that generate symbols for versions 1–40 and all four error-correction levels, render them as PNG, SVG or PDF, apply commercial styling (linear gradient, rounded modules, rounded frame, circular finder patterns, centre logo), and append an application-defined metadata trailer to the encoded data.

Correctness of such code is normally established by scanning a rendered image. That is a weak oracle for three reasons. First, Reed–Solomon decoding corrects up to t erroneous code words, where the number of error-correction code words is $2t$ Lin and Costello (2004); a symbol whose error-correction bytes are computed incorrectly can therefore still decode, because the decoder treats the discrepancy as channel noise. Second, the image is not the encoder: styling and rasterization change the image while leaving the code words untouched, so an image test conflates two layers. Third, decoders differ in what they tolerate and in how they interpret payload bytes Kieseberg et al. (2010). A scan test samples one decoder, one image and one input.

The input space is much larger than a test suite usually covers: 40 versions, four error-correction levels, eight masks, several segmentation choices and an arbitrary payload. The committed CQRC tests already compare seven python-qrcode-generated vectors at the module level, which is the right unit, but seven inputs cannot cover this space. Both defects reported below survive those tests.

Contributions.

1. *A dual implementation and its measured equivalence.* Two encoders with disjoint code paths (about 2,000 lines of C#, 837 lines of JavaScript, no third-party packages) agree bit-for-bit on 192 of 200 seeded inputs (1,164,776 modules), on 60/60 third-party oracle cases (585,988 modules), on all seven committed golden vectors, on a 40×4 structural sweep, and on a 160-point capacity check against four sources.
2. *A module-matrix differential method.* The conformance

unit is the tuple (version, mask, $W \times W$ matrix), with the interleaved code-word sequence as a diagnostic. Three oracles are combined: cross-language comparison, an external encoder invoked with an explicit mask, and an independent algebraic Reed–Solomon remainder check. We separate *policy compatibility* (same matrix as a reference encoder) from *invariant conformance* (properties the standard fixes), because QR encoding is not unique.

3. *Two defects, diagnosed.* The C# encoder crashes on any Reed–Solomon block whose remainder begins with a zero byte, because a polynomial indexer throws on negative indices; the failure occurs in 8 of 200 inputs and is predicted with accuracy 1.000. The C# PNG writer emits a zlib stream without the Adler-32 trailer, which both CPython and Node reject.

4. *Metadata insertion, measured and audited.* A trailer of four header bytes, a length and sorted-key JSON is appended as a normal byte-mode segment. It was recovered byte-for-byte from the de-interleaved code words in 20/20 cases, with the original text intact as the prefix, but the "ISO/IEC 21570" name is a misattribution: ISO 21570 is a food-analysis standard, not a QR standard.

5. *The geometric cost of styling.* Circular finders replace 16 of the 49 cells of each finder and remove 21.5% of its dark area while preserving the 1:1:3:1:1 profile along the axes; rounded modules remove up to 40% of a module's area; a light gradient endpoint has 1.80:1 contrast. Decodability is not claimed: no barcode decoder was available on the measurement host.

2 Background

2.1 Encoding pipeline

A Model 2 symbol of version $v \in [1, 40]$ is a square grid of $W = 17+4v$ modules. Finder patterns, separators, timing patterns, alignment patterns (for $v \geq 2$), the dark module and the reserved format areas are fixed; the remaining modules carry data and error-correction code words. Version information is added for $v \geq 7$.

Data is split into segments, one per mode: numeric (10 bits per three digits), alphanumeric (11 bits per two characters), byte (8 bits per byte) and kanji (13 bits per character). Each segment starts with a 4-bit mode indicator and a character-count field whose width depends on the mode and on the version group (1–9, 10–26, 27–40). The bit stream is terminated, padded to a byte boundary and filled with the alternating pad bytes `0xEC` and `0x11` up to the data capacity of the chosen version and level ISO/IEC (2015).

The capacity is expressed as a table of blocks: each of the 160 (version, level) pairs specifies a number of Reed–Solomon blocks, each with a data count and an error-correction count. For each block, the encoder computes the remainder of the message polynomial $m(x)$, shifted by n positions, modulo the generator polynomial

$$g(x) = \prod_{i=0}^{n-1} \left(x - \alpha^i\right) \tag{1}$$

over $\mathrm{GF}(2^8)$ with primitive polynomial `0x11D` Reed and Solomon (1960); Lin and Costello (2004). The data and error-correction code words of all blocks are interleaved, and the resulting bit stream is placed in two-module-wide vertical strips in a zigzag from the bottom right, skipping function patterns and the vertical timing column.

Eight mask patterns are then applied to the data modules, and the pattern with the lowest penalty under the four ISO scoring rules is selected; the format information (BCH(15,5), XOR mask `0x5412`) and, where applicable, the version information (BCH(18,6)) are written last.

2.2 What the standard fixes and what it leaves free

The standard fixes the function patterns, the format and version encoding, the Reed–Solomon code, the padding bytes and the placement. It does not fix the segmentation (multiple segmentations can encode the same text) and it does not require a particular mask: the penalty rules are guidance for choosing one. Two conforming encoders can therefore emit different matrices for the same input. This distinction matters for evaluation. We call *policy compatibility* the case where two encoders make the same segmentation and mask choices and hence produce the same matrix; we call *invariant conformance* the case where a symbol satisfies properties that the standard does fix (grid size, function patterns, valid BCH fields, Reed–Solomon divisibility, declared capacity). The method in Section 4 checks both, and does not present the first as the second.

3 Architecture

3.1 Two implementations

The C# library contains 19 source files (1,995 lines): a core encoder of 1,285 lines (bit buffer, segment splitting, block table, $\mathrm{GF}(2^8)$ arithmetic, Reed–Solomon, BCH, matrix construction, mask scoring), a 63-line metadata builder and a 647-line rendering layer. The JavaScript port is a single 837-line UMD file with the same pipeline; it can run in a browser or in Node.

Both implementations duplicate the same constant tables: the 160-row block table, the alignment-pattern coordinates and the character-count widths. The tables are literals, not generated at run time, so a transcription error is possible; we parsed the block table out of both sources and compared it element by element (160 rows, identical). The bit stream, padding, interleaving, placement and format fields follow the same code structure, but the code paths are separate: the ports share no source, and they differ in language semantics that matter here (integer division, array bounds, UTF-8 encoding, hashing).

Mask scoring deliberately reproduces the implementation choices of python-qrcode 8.2, because the committed golden

vectors were produced by that package: the 2×2 rule skips a column after a mismatch, the 1:1:3:1:1 rule uses an 11-module window with a one-position skip, rule 4 uses a floating-point dark ratio, and format bits are left light while scoring, matching python-qrcode's test mode. These choices are compatibility features, not claims of ISO conformance; Section 8 returns to this point.

The committed test suites are small. `node js/test.js` runs 16 checks (SHA-256, Reed–Solomon, structure and the seven golden vectors); `dotnet test` reports 15 passing xUnit results, of which 14 assert behaviour (seven code-word and seven matrix comparisons) and one is an empty placeholder. Both suites pass in this environment, as recorded in Table 2.

3.2 Rendering and styling

The matrix is rendered by a pixel loop over an output image that includes the four-module quiet zone. The renderer supports a solid or linearly interpolated foreground, a rounded outer frame, rounded module corners, circular finder patterns and a centre logo. PNG output is written by hand: IHDR, a single IDAT and IEND chunks, a hand-written CRC-32, a two-byte zlib header and a `DeflateStream` for the compressed payload. SVG and a one-page vector PDF are produced by the other renderers; the JavaScript port emits SVG and canvas images.

Styling is a pure function of the matrix: it changes colours and the shapes of cells but never the code words or the module values stored in the matrix. Any loss of decodability is therefore an image-level property, which is why Section 6 measures geometry rather than encoding.

4 Differential Testing at Module Level

4.1 Conformance unit and oracle protocol

For an input I (text, level, optional version, segmentation flag, optional metadata) an encoder returns a version v, a mask m, a matrix $M \in \{0,1\}^{W \times W}$ and an interleaved code-word sequence C. We treat (v, m, M) as the conformance unit and compare all W^2 modules, using a SHA-256 digest of the row-major matrix for speed and falling back to a full matrix diff to localize a mismatch. C is retained as a diagnostic: if two matrices differ, the code-word digest says whether the difference originates before or after placement.

Three oracles are used.

- *Cross-language.* The same input is encoded by both implementations and the matrices are compared. This detects divergence but cannot decide which side is right.
- *External encoder.* python-qrcode 8.2 is constructed as `QRCode(version, ec, border=0)`, fed with `add_data(text, optimize=4)` (or `optimize=0`), fitted, and queried for its best mask; `makeImpl(False, mask)` then returns the matrix for that explicit mask. Passing the mask explicitly removes mask selection from the oracle, so a disagreement is attributable to encoding or placement rather than to scoring. The oracle shares no code with either implementation.
- *Algebraic invariant.* The harness de-interleaves the code words into per-block data and error-correction bytes and computes, in Python, the remainder of the concatenation $\texttt{data}\|\texttt{ec}$ modulo $g(x)$ over $\mathrm{GF}(2^8)$. Every block must have a zero remainder. This check is independent of both encoders and of python-qrcode, and it validates error correction even for symbols whose matrix is not compared.

The matrix unit also makes failures observable that an image test hides: a wrong error-correction byte changes modules, but a decoder may correct it and report success.

Family	Cases	Modules compared
Fixed version (one per version)	40	477,320
Random auto-version	100	422,908
Metadata trailer	20	70,604
Capacity boundary	20	168,324
Optimization-sensitive	20	25,620
Total	200	1,164,776

Table 1: Seeded differential corpus. Modules are counted only for the 192 inputs for which both implementations returned a matrix; the eight C# failures have no matrix to compare. Versions 1–40 all occur.

4.2 Corpus

The corpus is generated by a Python script with seed 20261021 and contains 200 cases in five families (Table 1). Fixed-version cases force each version once, cycling the level; random cases draw payloads from six generators (numeric, alphanumeric, URL, ASCII, UTF-8 with CJK and emoji, and mixed text with digit runs) with sizes log-uniform between 2 and about 1,600 bytes; metadata cases add one to three alphanumeric keys and randomly enable the automatic content hash; boundary cases use a byte payload of exactly the capacity or one byte less for a random (version, level); optimization cases embed digit runs in prose so that greedy segmentation changes the result. In total 94 cases exercise the segmentation optimizer, and all 40 versions occur.

4.3 Structural and capacity sweeps

Two exhaustive checks complement the random corpus. The structural sweep forces each of the 160 (version, level) pairs with a short payload and verifies the grid size $17 + 4v$, a mask in $0 \ldots 7$, the expected code-word count and the absence of unfilled data modules; both implementations pass all 160 combinations. With this fixed short payload the penalty function selects mask 4 in 80 of the 160 combinations and never selects mask 7 (histogram: 35, 8, 28, 3, 80, 2, 4, 0 for masks 0–7). That distribution is a property of the payload, not of the scorer, and is reported to avoid the impression that mask 4 is canonical.

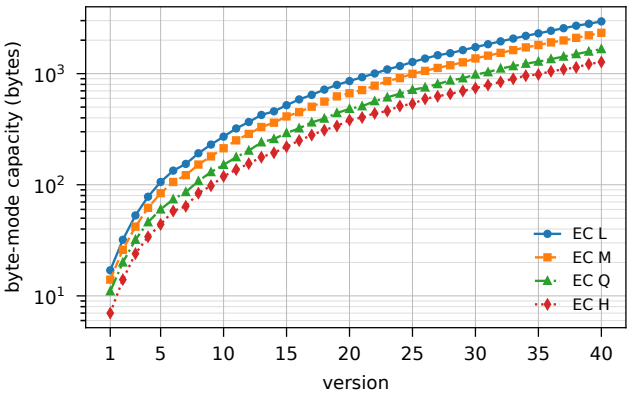


Figure 1: Byte-mode capacity returned by the binary search for all 40 versions and four levels (160 points, logarithmic ordinate). Every value agrees with the C# search where computed (28 points), with python-qrcode 8.2 (64), with transcribed ISO/IEC 18004 Table 7 values (44), and with the block-table capacity (all 160).

Check	Cases	Unit	Result
Committed JS suite	16	tests	16 pass
Committed C# suite	15	tests	15 pass[a]
Golden vectors	7	11,375 modules	7/7 both
Structural sweep	160	(version, level)	160/160 both
Capacity	160	byte count	0 mismatch
Cross-language	200	1,164,776 modules	192 equal, 8 throws
python-qrcode oracle	60	585,988 modules	60/60
Metadata trailer	20	8 checks each	20/20
PNG writer	1	IDAT / zlib	invalid stream

Table 2: All checks performed. [a]14 of the 15 assert behaviour (seven code-word and seven matrix comparisons); one is an empty placeholder counted by xUnit.

Capacity is checked by binary search on the length of a forced byte-mode payload, at a fixed version, in both implementations. The resulting 160 values are compared with four sources: the C# search at 28 points (versions 1, 2, 3, 5, 10, 20 and 40), python-qrcode's behavioural fit at 64 points (16 versions), the published ISO/IEC 18004 Table 7 values at 44 points (versions 1–10 and 40), and the capacity implied by the block table at all 160 points. There are no mismatches: for example version 1/H holds 7 bytes, version 10/M holds 213 and version 40/L holds 2,953. Figure 1 shows the full surface.

4.4 Results

Table 2 collects every check. The committed suites and the seven golden vectors pass. The structural sweep passes in both languages and the two implementations choose the same mask in all 160 cases. Capacity has no disagreement across four sources. The cross-language corpus is bit-exact on all 192 inputs that both sides accept, and the external oracle agrees on all 60 cases it was given. The two failures are the exceptions: eight inputs crash the C# encoder, and the PNG writer produces a stream that no standard zlib decoder accepts.

4.5 Defect 1: a negative index in the encoder

The eight failing inputs all terminate with the same message, `Index was outside the bounds of the array`, and no matrix is produced. The JavaScript port accepts the same inputs, and an independent Reed–Solomon check finds a zero remainder for every block of every JavaScript output, including these eight.

The cause is in the interaction of two helpers. `GfPoly` stores coefficients most-significant first and strips leading zeros, so a remainder with z leading zero bytes has length $n_{\mathrm{ec}} - z$. Its indexer (`GfPoly.cs:25`) returns 0 for an index beyond the stored coefficients but throws for a negative index. `ReedSolomon.Encode` (`ReedSolomon.cs:31–33`) right-aligns the remainder in the n_{ec}-byte field by setting `offset = mod.Length - ecCount` and copying `mod[i + offset]` for $i = 0, \ldots, n_{\mathrm{ec}} - 1$. When $z > 0$, `offset` is negative and the first z reads are out of range. C# throws; JavaScript evaluates the same reads to `undefined`, which becomes zero when stored into the code-word buffer. Those zeros are precisely the padding the field requires, so the JavaScript result is correct.

The hypothesis predicts exactly which inputs fail: a symbol fails if and only if at least one of its Reed–Solomon blocks has a remainder whose first byte is zero. Over the 200 inputs there are 2,292 blocks and nine such remainders, in eight symbols (one symbol has two). The predictor has 8 true positives, 0 false positives, 0 false negatives and 192 true negatives, for accuracy 1.000. Under a uniform first-byte model one expects $2{,}292/256 \approx 8.95$ zero-leading remainders; nine were observed. The smallest failing case is a version-2 symbol with a single block (`auto-043-alnum`). The defect is therefore not a rare corner of the specification but an input-dependent crash with probability around 4% for random payloads of this corpus, absent from the seven golden vectors because none of their blocks happened to start with a zero byte.

4.6 Defect 2: a PNG stream without Adler-32

The C# renderer writes a 328×328 RGBA image for a version-4 symbol. The container is well formed: IHDR, one IDAT and IEND are present, all three CRC-32 values are correct, the zlib header is `78 9C`, and the 5,458 raw DEFLATE bytes inflate to exactly 430,664 bytes (328×328×4 plus 328 filter bytes). The stream nevertheless has no Adler-32 trailer: the last four bytes are `F5 EF FC 1F`, whereas the Adler-32 of the inflated payload is `F0 EA 7C F8`. CPython's zlib reports "incomplete or truncated stream" and Node's zlib reports "unexpected end of file". The code comment above the compressor states that `DeflateStream` supplies the Adler-32; it does not, because it emits raw DEFLATE rather than a zlib wrapper. A test that checks only chunk CRCs, as the project's tests do, cannot see this.

The two findings are complementary. The first is an input-dependent logic error that a small set of golden vectors misses;

the second is a systematic format error that a container-level check misses. Neither is detected by the committed suites, and both are reproducible from the scripts in the artifact.

5 Auxiliary Metadata after the Data

5.1 Wire format

The project appends optional metadata after the user's text. The metadata is first serialized as a JSON object with keys sorted by ordinal comparison and values as strings (or `null`); when enabled, a `content_hash` field is added with the SHA-256 of the concatenated input, in lowercase hex. The serialized bytes are placed behind a six-byte prefix:

$$\texttt{00 00 A1 00} \,\|\, \mathrm{L}_{16} \,\|\, \text{JSON},$$

where L_{16} is the JSON length as a 16-bit big-endian integer. The whole record is then wrapped in a normal byte-mode segment (4-bit mode indicator, character-count field, payload) and appended to the segment list before encoding. It is therefore protected by the same Reed–Solomon code as the text, and it sits at the end of the data stream. No structured-append symbol, ECI designator or new error-correction level is involved.

In the 20 metadata cases the JSON ranged from 14 to 134 bytes and the whole segment from 172 to 1,140 bits. For versions 1–9 the byte-mode count field is eight bits and for versions 10–40 it is sixteen; the largest payload observed was 140 bytes, so the field is not exceeded. A payload above 255 bytes could not fit a version-9 symbol in any case (version 9/L holds 232 bytes), so the eight-bit field is sufficient by construction in this format.

5.2 Round-trip from the code words

We de-interleaved the data code words of each metadata symbol, parsed the segment stream (numeric, alphanumeric and byte modes, with the version-dependent count widths) and checked eight properties per case: the metadata record is the last segment; the stream is terminated or exactly fills the capacity; the four header bytes match; the 16-bit length equals the remaining payload length; the length fits the count field; the JSON parses and equals the expected object (including the content hash); the keys are sorted; and the text segments that precede the record reproduce the original input exactly. All 20 cases pass all eight checks.

Two consequences follow. The record is recoverable from the transmitted code words alone, so a decoder or post-processor can separate it from the text; and the text is not displaced, because it occupies the earlier segments unchanged.

5.3 Decoder semantics and label provenance

ISO/IEC 18004 decoders concatenate the payloads of all segments in order. A conforming decoder therefore returns the original text followed by the six-byte prefix and the JSON, not the original text alone. The prefix bytes are not valid UTF-8, so the exact decoded string depends on the decoder's byte-to-text policy (stop at a NUL, substitute replacement characters, or expose bytes). An application that compares the decoded string with the input will observe a mismatch. The trailer is best understood as application data that a metadata-aware consumer strips, not as transparent metadata that other decoders ignore.

The name "ISO/IEC 21570" is not supported. ISO 21570:2005 is *Foodstuffs—Methods of analysis for the detection of genetically modified organisms and derived products—Quantitative nucleic acid based methods* ISO (2005), a 110-page food-analysis standard with no relation to barcodes; no ISO/IEC 21570 exists in the AIDC committee's programme. The header constant `00 00 A1 00` is private to this project. Calling the record "ISO/IEC 21570 metadata" in documentation, as the repository does, invites interoperability assumptions that no third party can satisfy. If a registered syntax is required, ISO/IEC 15434 defines a syntax for high-capacity ADC media ISO/IEC (2006); otherwise the record should be documented as a private extension with its own version field.

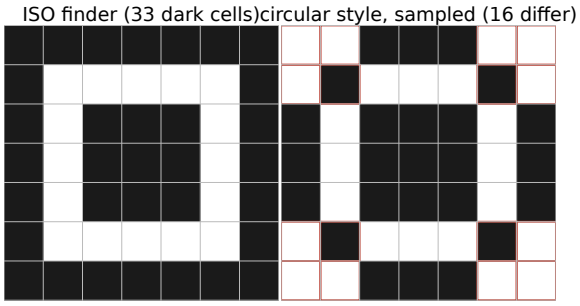


Figure 2: Left: the ISO 7×7 finder (33 dark cells). Right: the circular replacement sampled at module centres; red outlines mark the 16 cells whose value changes. The dark/light/dark/-dark/dark/light/dark profile along both axes, i.e. the 1:1:3:1:1 ratio, is preserved.

6 The Geometric Cost of Styling

The circular-finder option replaces each 7×7 finder pattern by a filled disc of radius 1.5 module units and a ring between radii 2.5 and 3.5, evaluated per output pixel. Sampling the result at module centres, 16 of the 49 cells change: the four 2×2 corner blocks of the standard pattern become light, while the 33 cells of the cross and the centre remain dark (Figure 2). In a version-3 symbol the three finders contribute 48 changed cells out of 841 modules, or 5.7% of the matrix.

The dark area of a finder falls from 33 to $8.25\pi \approx 25.92$ module squares, a loss of 21.5% (from 67.3% to 52.9% of the 7×7 box). A horizontal or vertical scan through the centre still crosses dark runs of 1, 3 and 1 modules separated by light runs of 1, as the standard requires. The pattern is thus geometrically plausible, but the corner samples—which some decoders use when locating a symbol—are inverted.

Feature	Measure	Value
Circular finder	cells changed per finder	16/49
	cells changed per version-3 symbol	48/841
	dark area (module^2)	$33 \rightarrow 25.92$
	axial dark runs	1:3:1
Module rounding	area removed at 0.1/0.25/0.3/0.5	4/12/20/40%
Gradient	contrast at `#1A1A1A`/`#0066CC`/`#66CCFF`	17.4/5.57/1.80
Logo	area at default width	2.25% (not tested)

Table 3: Measured geometry of the styling options. None of these measurements is a scan test.

Scenario	Payload (bytes)	JS (enc/s)	C# (enc/s)	C#/JS
URL, auto version	49	329.1	440.0	1.34
URL, fixed version 10	49	107.2	133.6	1.25
URL, auto, hash	49	146.7	173.5	1.18
Numeric, auto version	40	709.2	910.5	1.28

Table 4: Encode throughput, median of five runs. "enc/s" is complete `generate()` calls per second; the reciprocal is the cost per symbol.

Rounded modules remove corner pixels from every dark cell. At a rounding ratio of 0.1, 0.25, 0.3 and 0.5 (with a 10-pixel module) the removed area is 4%, 12%, 20% and 40% of the cell, and the module centre remains inside the drawn shape in all four settings. At the maximum ratio the module is visibly a rounded blob; the code words are unchanged, but the fraction of ink available to a binarizer falls sharply.

Gradient colouring introduces a contrast limit at one end of the symbol. The default foreground `#1A1A1A` on white has a contrast ratio of 17.4:1; `#0066CC` has 5.57:1; `#66CCFF` has 1.80:1, below the 3:1 threshold that WCAG 2.1 applies to non-text content W3C (2018). A gradient from the second to the third colour therefore ends in a region where the symbol's modulation is weak. The default logo occupies 15% of the symbol width, or 2.25% of its area before padding; we did not evaluate logo coverage. These numbers describe the image, not a scanning outcome.

No barcode decoder was installed on the host: OpenCV, pyzbar and zbarimg were absent, and only Pillow was available. The repository's README states that the circular style has been read by jsqr and by phone cameras; we did not reproduce that claim, and we make no decodability claim of our own. The measurements above identify where a decoder would have to cope: inverted corner cells in the finders, up to 40% less ink per rounded module, and a low-contrast gradient end.

7 Encoding Cost and Toolchain

All measurements ran on one Windows host with .NET SDK 10.0.402 (runtime 10.0.13), Node v24.21.0 and Python 3.13.14. The qrcode package (8.2) was already installed; no package was installed from the network, and the version was read from the installed metadata. The committed suites ran verbatim: the JavaScript suite passed 16/16 in 0.47 s and the C# suite reported 15/15 in 8.52 s, including the empty placeholder discussed above.

Table 4 reports encode throughput as the median of five runs of 100 or 300 iterations per scenario, after warm-up, on a single thread. The C# implementation is 1.18–1.34 times faster than the JavaScript one on these scenarios. A fixed version-10 symbol costs about three times a version-4 URL symbol, as expected from the growth of the matrix and the eight mask trials. Enabling the automatic content hash raises the cost of the version-4 URL case by about 2.2 times in JavaScript and 2.0 times in C#, which is consistent with SHA-256 over the payload plus JSON serialization. These are host-specific figures from a non-isolated process; they are useful for comparing the two implementations, not for comparing languages.

The toolchain also determines what could be tested. .NET was available, so the C# suite and the C# harness ran; the numbers in this paper are not inferred from the JavaScript side alone. Python was available, so python-qrcode could serve as an external oracle and the independent Reed–Solomon check could be written. No barcode decoder was available, so the styling study stops at geometry.

8 Threats to Validity

Independence. The two implementations were written by the same author and share tables and design decisions. Their agreement is evidence of consistency, not of independent derivation. The external oracle and the algebraic check are the strongest defences against a shared misunderstanding; both are limited to the cases they cover.

The oracle is an implementation, not the standard. python-qrcode is a widely used encoder, but it is not a normative reference. The CQRC mask scorer deliberately copies its penalty quirks, so the 60/60 oracle result demonstrates compatibility with python-qrcode, not conformance to ISO/IEC 18004. The structural sweep and the capacity check are the parts of the evaluation that test standard-fixed properties; the mask histogram and the golden vectors are compatibility results.

Sampling. The corpus has one seed and 200 cases. The eight failures are input-dependent, and a different seed would produce a different count; the per-block probability of a zero-leading remainder is about $1/256$, and the observed nine in 2,292 blocks is consistent with it. The 60 oracle cases cover versions 1–40 with one fixed case per version and 20 random cases; they do not cover every mode/level combination.

Capacity sources. Only 44 of the 160 capacity values were compared with transcribed ISO table entries (versions 1–10 and 40), and the codeword-derived value uses the same block table that both implementations use, so it is not an independent oracle. The behavioural and C# searches are independent code paths but share the table.

Metadata. The 20 metadata cases exercise the project's private trailer, not any external standard. No third-party consumer was available to parse it, and the round-trip check uses our own parser. The claim that a conforming decoder returns text plus trailer follows from the segment semantics, but it was not observed with an external decoder.

Styling and scanning. No decoder and no camera were available. The styling results are geometric measurements; the repository's README claim about jsqr and phone cameras is not reproduced. Contrast ratios are computed from the configured colours, not measured from a rendered image or a print.

Read-only repository. Both defects remain in the repository. No regression test was added, and the C# suite still passes. The artifact reproduces the failures but does not fix them.

9 Related Work

Standards and encoders. ISO/IEC 18004 defines the symbology, including the block table and the mask penalty rules ISO/IEC (2015); Reed–Solomon coding and its algebraic decoding are classical Reed and Solomon (1960); Lin and Costello (2004). Open-source encoders such as python-qrcode Lincoln Loop (2025) are the de facto reference for many projects, and their choices (segmentation, mask scoring, padding) become compatibility targets. CQRC is unusual in maintaining two language ports with a shared golden-vector suite and in testing the matrix rather than the image.

Differential testing and oracles. Differential testing compares implementations on the same input McKeeman (1998); it found many compiler bugs when applied at scale Yang et al. (2011). Random generation of inputs is the standard way to widen coverage Claessen and Hughes (2000). The oracle problem—what to compare against when no reference exists—is surveyed by Barr et al. Barr et al. (2015); metamorphic relations and algebraic invariants are one answer Chen et al. (2018). Our Reed–Solomon divisibility check is such an invariant, and the explicit-mask oracle is an attempt to isolate one stage of the pipeline.

Styling. Halftone QR codes replace modules by sub-module patterns to embed an image while preserving a readable core Chu et al. (2013); QArt codes use the error-correction capacity to paint a picture in the data region Cox (2012). Both accept a decodability risk that is usually assessed by scanning. Our contribution is narrower: we quantify the geometric changes of a commercial styling layer and state explicitly that scanning was not measured.

Security and interpretation. Kieseberg et al. analyse the security implications of QR payloads and the role of decoder behaviour Kieseberg et al. (2010); the metadata discussion in Section 5 is a related interoperability caution: a private trailer changes the decoded string even though the symbol remains valid.

10 Conclusion

CQRC is a small artifact that exposes a general problem: QR encoders are easy to write and hard to validate with images. Treating the module matrix as the conformance unit, and adding an external encoder and an algebraic Reed–Solomon check as oracles, produced a precise account of two implementations: 192 of 200 inputs are bit-identical, 60/60 oracle cases agree, all 160 structural combinations pass, and capacity agrees at all 160 points. The same method found two defects that the committed suites missed: a C# crash on Reed–Solomon blocks with a zero-leading remainder, predicted with accuracy 1.000 from the code words, and a PNG IDAT stream without the Adler-32 trailer, rejected by two independent zlib implementations.

The study also documents two limits that are easy to overlook. A metadata record appended as a byte segment is recoverable and leaves the text intact, but a standard decoder returns text plus record, and the project's "ISO/IEC 21570" label is a misattribution to a food-analysis standard. Styling changes the image in measurable ways—16 of 49 finder cells, up to 40% of a module's area—and those changes were not tested against a decoder. Future work should add a decoder oracle (a software reader plus a print-and-scan study), fix the two defects, and turn the leading-zero predictor into a regression test.

References

E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525, 2015. doi: 10.1109/TSE.2014.2372785.

T. Y. Chen, F.-C. Kuo, H. Liu, et al. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4:1–4:27, 2018. doi: 10.1145/3143561.

H.-K. Chu, C.-S. Chang, R.-R. Lee, and N. J. Mitra. Halftone QR codes. *ACM Transactions on Graphics*, 32(6):217:1–217:8, 2013. doi: 10.1145/2508363.2508408.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000. doi: 10.1145/351240.351266.

R. Cox. QArt codes. `https://research.swtch.com/qart`, 2012.

ISO. ISO 21570:2005: Foodstuffs — methods of analysis for the detection of genetically modified organisms and derived products — quantitative nucleic acid based methods. International Organization for Standardization, Geneva, 2005.

ISO/IEC. ISO/IEC 15434:2006: Information technology — automatic identification and data capture techniques — syntax for high-capacity ADC media. International Organization for Standardization, Geneva, 2006.

ISO/IEC. ISO/IEC 18004:2015: Information technology — automatic identification and data capture techniques — QR code bar code symbology specification. International Organization for Standardization, Geneva, 2015.

P. Kieseberg, M. Leithner, M. Mulazzani, et al. QR code security. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia (MoMM)*, pages 430–435. ACM, 2010.

S. Lin and D. J. Costello. *Error Control Coding*. Pearson Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2004.

Lincoln Loop. python-qrcode: QR code image generator. https://github.com/lincolnloop/python-qrcode, 2025. Version 8.2, released 1 May 2025.

W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. doi: 10.1137/0108018.

W3C. Web content accessibility guidelines (WCAG) 2.1. https://www.w3.org/TR/WCAG21/, 2018.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011. doi: 10.1145/1993498.1993532.