ABSTRACT

Introduced in 1991, libjpeg has become a well-established library for processing JPEG images. Many libraries in high-level languages use libjpeg under the hood. So far, little attention has been paid to the fact that different versions of the library produce different outputs for the same input. This may have implications on security-related applications, such as image forensics or steganalysis, where evidence is generated by tracking small, imperceptible changes in JPEG-compressed signals. This paper systematically analyses all libjpeg versions since 1998, including the forked libjpeg–turbo (in its latest version). It compares the outputs of compression and decompression operations for a range of parameter settings. We identify up to three distinct behaviors for compression and up to six for decompression.

1 INTRODUCTION

JPEG is a popular method for image compression that reduces imperceptible information (and hence file size) while preserving perceptual quality. Standardized in 1991, the method has been implemented in many codecs, and the data format is supported by thousands of applications [12]. Due to its popularity, extensive research covers many aspects of JPEG compression [9, 10, 21].

The JPEG standard does not prescribe a single canonical way of compression and decompressions. This “generic” approach [31] leaves some degree of freedom for different implementations to be tailored to the needs of the application or platform. As a result, not every instance of a JPEG codec produces the same output for a given input. While small numerical deviations may not be perceptible to human observers and thus not relevant in the majority of applications, a number of applications in multimedia security rely on subtleties of compression and decompression operations.

Chiefly these include image forensics and steganalysis. A common technique in image forensics is to analyze JPEG compression artifacts to estimate the compression history or detect manipulations of a digital image [32]. Steganalysis aims to detect the existence of steganographic messages. Recent machine learning-based steganalysis has been reported to be sensitive to the exact knowledge of the cover generation process, which includes JPEG pre-compression [11]. Moreover, a well-known steganalysis technique is the JPEG compatibility test [10], which detects embedding changes in stego objects created from JPEG pre-compressed covers. A recent revision of this technique has pointed out the sensitivity to the decompressor implementation [8], but a systematic evaluation of the differences between libjpeg versions is still missing.

The effect of the JPEG implementation has been subject of prior research, but existing results are fragmented and usually compare pairs of implementations only. Observing such differences is quite straightforward, as illustrated in Figure 1. A never-compressed source image has been compressed using 4:2:0 color subsampling.
According to the IJG, version 6b, is the first “solid and stable version” [13]. It has not been updated for more than a decade, making it the de-facto standard JPEG codec for image processing libraries [28]. In 2010, libjpeg-turbo was introduced, which optimizes the performance of version 6b by using SIMD instructions: MMX and SSE on x86, and later Neon on ARM [1]. This strand has seen 35 different versions between 2010 and 2021, which we deemed too many to include all of them in this analysis. Instead, we chose to use libjpeg-turbo version 2.1.0. In 2014, Mozilla forked libjpeg-turbo into mozjpeg, which reduces web page loading times by enabling stronger compression at the same perceived quality. This leads to shorter output sizes as lossless compression, namely difference encoding followed by run-length and Huffman coding, is applied when producing the bit stream. JPEG decompression reconstructs this process to the extent possible with the remaining information. The most relevant parameters exposed by the libjpeg API are:

- **Color Space.** Spatial domain images are typically represented in RGB colors or as grayscale. The first step of JPEG compression maps colors to the \( YC_bC_r \) space, where the \( Y \) channel describes the luminance of a pixel, and \( C_b \) and \( C_r \) represent its color information, called chrominance. Grayscale images have the \( Y \) channel only. Version 8d of libjpeg introduces the option to store RGB channels directly in a JPEG image, called RGB JPEG.

- **Chroma Subsampling.** Since the human eye is more sensitive to changes in brightness than to changes in color, compression can be increased by storing chrominance values at lower resolution. This is achieved by combining larger quantization factors with the option to reduce the resolution of the chrominance channels prior to the block-wise DCT. The reduction in resolution is controlled by the subsampling factor, for which different notations exist. Let

and then decompressed using “fancy” upsampling, once with libjpeg version 6b and once with version 9e. The two resulting output images are compared in the spatial domain with differences amplified by setting the respective RGB value to maximum intensity (255). Lai and Böhme [18] note that the choice of libjpeg’s DCT method affects the reliability of a forensic technique based on JPEG block converges in grayscale images. Carnein et al. [5] generalize this method to color and draw attention to differences in the color subsampling method used in libjpeg version 6 and 7. Bonettini et al. [4] show how to distinguish two JPEG implementations from their outputs: Adobe’s proprietary one and the Python Imaging Library (PIL) version 5.2.0, which uses libjpeg under the hood. Their method is claimed robust to the choice of the quantization matrix and the presence of double compression. Most recently, Lorch and Riess [20] discover the characteristic “chroma wrinkles,” which are caused by the rounding to integers in libjpeg version 9a (when set to simple scaling) and libjpeg-turbo version 2.0.1 (by default).

To the best of our knowledge, this work contributes

- the first systematic comparison of all libjpeg versions since 1998, including the recent version of libjpeg-turbo (2.1.0);
- results from two series of experiments, one for compression and one for decompression;
- an analysis of the impact of parameters supported by the libjpeg API on differences between versions; and
- a quantification of the magnitude of the observed differences.

This paper proceeds as follows. Section 2 reviews the evolution of libjpeg and recalls relevant parameters of its API. Section 3 documents the experiments and all results. Section 4 discusses the findings, describes limitations, and outlines future work.

## 2 BACKGROUND

In 1991, the Joint Photographic Experts Group (JPEG) submitted the specification for the JPEG format, which was later approved as an international standard for image processing by the ISO [14] and ITU [15]. It was implemented by the Independent JPEG Group (IJG), as an open-source, free and easy-to-use C library, called *libjpeg* [13].

### 2.1 Evolution of Libjpeg

According to the IJG, version 6b, is the first “solid and stable version” [13]. It has not been updated for more than a decade, making it the de-facto standard JPEG codec for image processing libraries [28]. In 2010, *libjpeg-turbo* was introduced, which optimizes the performance of version 6b by using SIMD instructions: MMX and SSE on x86, and later Neon on ARM [1]. This strand has seen 35 different versions between 2010 and 2021, which we deemed too many to include all of them in this analysis. Instead, we chose to use libjpeg-turbo version 2.1.0 only. In 2014, Mozilla forked libjpeg-turbo into *mozjpeg*, which reduces web page loading times by enabling stronger compression at the same perceived quality. This optimization comes at the cost of compression performance [22]. Mozjpeg uses progressive JPEG by default, unlike all versions considered in this work. As any comparison would be artificial and hard to interpret, we exclude mozjpeg from our analysis. In 2019, libjpeg-turbo became the official reference implementation for the JPEG standard [16]. Figure 2 illustrates the timeline of all libjpeg versions and relevant forks between 1998 and 2022, and Table 1 summarizes relevant changes to the library. Some terminology in this table is specific to the libjpeg API and introduced next.

### Table 1: Authors’ digest of the libjpeg release notes [24]

<table>
<thead>
<tr>
<th>Version</th>
<th>Release</th>
<th>Significant changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2009-06-27</td>
<td>DCT scaling; arithmetic coding; color interpolation (fancy downsampling)*</td>
</tr>
<tr>
<td>8</td>
<td>2010-01-10</td>
<td>SmartScale extension</td>
</tr>
<tr>
<td>8a</td>
<td>2010-02-28</td>
<td>Improve accuracy in floating point IDCT</td>
</tr>
<tr>
<td>8b</td>
<td>2010-05-16</td>
<td>Minor improvements</td>
</tr>
<tr>
<td>8c</td>
<td>2011-01-16</td>
<td>Minor improvements</td>
</tr>
<tr>
<td>8d</td>
<td>2012-01-15</td>
<td>RGB JPEG file support</td>
</tr>
<tr>
<td>9</td>
<td>2013-01-13</td>
<td>Reversible color transform for RGB JPEG files (not backward compatible)</td>
</tr>
<tr>
<td>9a</td>
<td>2014-01-19</td>
<td>Wide gamut color spaces; more accurate color conversion; extended bit depth and entropy decoder</td>
</tr>
<tr>
<td>9b</td>
<td>2016-01-17</td>
<td>Improvements in DCT and color calculations</td>
</tr>
<tr>
<td>9c</td>
<td>2018-01-14</td>
<td>Minor improvements</td>
</tr>
<tr>
<td>9d</td>
<td>2020-01-12</td>
<td>Huffman code table generation optimization; 64-bit platform support</td>
</tr>
<tr>
<td>9e</td>
<td>2022-01-16</td>
<td>Change of default chrominance DC quantization factor (to support lossless)</td>
</tr>
</tbody>
</table>

* Complementing fancy upsampling implemented earlier

## 2.2 Relevant API Parameters

Recall that JPEG compression cuts a spatial domain image channel into blocks of (usually) 8 × 8 pixels, applies block-wise 2D discrete cosine transformation (DCT), and divides the resulting coefficients by subband-specific quantization factors before rounding to the nearest integer. This step is lossy with adjustable quality: larger quantization factors result in smaller numbers and more zeros. This leads to shorter output sizes as lossless compression, namely difference encoding followed by run-length and Huffman coding, is applied when producing the bit stream. JPEG decompression reconstructs this process to the extent possible with the remaining information.

The most relevant parameters exposed by the libjpeg API are:

- **Color Space.** Spatial domain images are typically represented in RGB colors or as grayscale. The first step of JPEG compression maps colors to the \( YC_bC_r \) space, where the \( Y \) channel describes the luminance of a pixel, and \( C_b \) and \( C_r \) represent its color information, called chrominance. Grayscale images have the \( Y \) channel only. Version 8d of libjpeg introduces the option to store RGB channels directly in a JPEG image, called RGB JPEG.

- **Chroma Subsampling.** Since the human eye is more sensitive to changes in brightness than to changes in color, compression can be increased by storing chrominance values at lower resolution. This is achieved by combining larger quantization factors with the option to reduce the resolution of the chrominance channels prior to the block-wise DCT. The reduction in resolution is controlled by the subsampling factor, for which different notations exist. Let
We set up and run a series of experiments to compare all libjpeg versions since 1998 up to 9e and the recent version 2.1.0 of libjpeg—indeed. To facilitate the comparison of the different versions we use a custom quality scale and the resulting JPEGs can be identified by their "non-standard" quantization matrices [25].

3 EXPERIMENTS

We use the datasets ALASKA2 [6] for color, and BOSSBase [2] for grayscale images. The never-compressed images are cropped to $512 \times 512$ (BOSSBase) and $256 \times 256$ (ALASKA2) pixels. We sample 1,000 images from each database.

We design compression and decompression experiments for each parameter, described in Section 2.2, as illustrated in Figure 3 and 5, respectively. For the compression experiments, we change the value of the investigated parameter systematically within a specified domain and compare the outputs after compressing with each version on the level of DCT coefficients. For the decompression experiments, we first compress an image with a fixed default version and then decompress it with all versions while controlling the parameters. Again, we compare the outputs generated by the different versions, however now in the spatial domain. We run all experiments using versions 6b, 7, and 9e as fixed default for compression and observe equivalent decompression results. This confirms that our choice of a specific version for compression does not affect the results.

DCT Method. Libjpeg implements three DCT methods for compression and inverse DCT for decompression, namely *ifast*, *islow*, and *float*. They differ in the data type of intermediate results, speed, and output quality. All methods use FDCT. The prefix F denotes the "fast" divide-and-conquer algorithm found in textbooks [30].

Quality. The quantization factors regulating the information loss are stored in quantization matrices. In principle, they can be chosen arbitrarily, but JPEG defines a scalar quality factor (QF) in the range between 1 and 100. Each integer indexes a pair of quantization matrices for luminance and chrominance, respectively. A QF of 100 sets all elements to 1, resulting in the best achievable quality.

Some implementations, most notably Adobe and ImageMagick, use a custom quality scale and the resulting JPEGs can be identified by their "non-standard" quantization matrices [25].

When comparing outputs, we flag a difference if at least one DCT coefficient (for compression) or pixel value (for decompression) in the output differs between two versions. To define a baseline for comparisons we use the following default parameter values: QF 75, DCT method *islow*, and fancy upsampling with factor 4:2:0.

Considering that saturated areas may be processed differently as the truncation to the value range involves a non-linear operation [3, 29], we rerun all our experiments on a set of special images. The set includes synthetic checkerboard images (both alighted and unaligned with JPEG blocks) as well as on the two images with the highest ratio of 0s (dark saturation) and 255s (white saturation) in each dataset. We find no peculiarity for the special images.

The code for our experiments is available on GitHub. It uses our Python package that allows users to select the libjpeg version.

Figure 3: System model for the compression experiments

Figure 2: Timeline of libjpeg versions and forks between 1998 and January 2022

1https://github.com/uibk-uncover/KnowYourLibrary

2The most saturated images are "10343.tif" (50.1 % dark saturation) and "05887.tif" (39.7 % white saturation) in ALASKA2, and "6155_6_1.png" (38.2 % dark saturation) and "6960_1_3.png" (100 % white saturation) in BOSSBase.
Figure 4: Differences between versions for compression

3.1 Compression

To measure the influence of the library version on compression, we first carry out baseline experiments with default parameters for every version. We then compare the results to those generated using different parameter values for chroma subsampling (simple and fancy subsampling and different subsampling factors), different DCT methods, and QFs. Figure 4 reports all mismatches observed.

**Baseline.** The compression with default parameters produces three sets of versions producing identical outputs: (1) libjpeg 6b and libjpeg–turbo, (2) libjpeg 7 up to 9d, and (3) libjpeg 9e. For grayscale images, all versions produce the same results. The observations indicate changes in libjpeg 7 and 9e regarding the processing of color. Hence, the next experiment analyses the version’s influence on color subsampling.

**Color Subsampling.** We test all subsampling factors in this list, (4:4:4, 4:4:0, 4:2:2, 4:2:0, 4:1:1, 4:1:0, 3:3:0, 3:3:1, 3:1:0, 2:1:0), and systematically vary between simple and fancy mode. To test uncommon but permitted inputs, we specify some of the factors in several equivalent ways, for example 4:4:4 (no subsampling) can also be stated as 3:3:3. The libjpeg API allows for even more redundant encoding as the subsampling is specified in six parameters. Finally, we compress images with subsampling in only one of the two chroma channels while keeping the other one at full resolution.

We find that if no subsampling or any kind of simple subsampling is used, the distinction between (1) 6b and turbo and (2) 7 to 9d disappears. If fancy subsampling is selected, this is also the case whenever no subsampling factor is a power of two. For example, 4:4:4 (no subsampling) and 3:3:3 (no power of two) produces outputs where no distinction between the versions 6b to 9d including turbo can be made. We denote this as 3××××× in Figure 4.

We suspect that this is caused by a fallback function for DCT scaling introduced in version 7, which exclusively supports powers of two. In other words, if the subsampling mode is fancy and any of the chroma channels is subsampled with a power of two, DCT scaling becomes effective, leading to differences in the outputs produced with (1) version 6b and turbo and (2) 7 to 9e. None of our exotic calls refutes this explanation.

**DCT Method.** We explore the influence of the libjpeg version on compression results produced with all three DCT methods.

We find that changing DCT methods results in the same differences as seen in the baseline; for grayscale and color, both with and without chroma subsampling. Note that the three DCT methods produce significantly different outputs for any fixed version.

**Quality.** We run our experiments for all QFs from 50 to 100. We observe baseline results (QF 75) for all QFs except 64, 67, 70, 73, 76, 78, 79, 81, 82, 84, 85, and 87–100. In those cases, versions 7 to 9d produce identical outputs as 9e. The differences are caused by the change of the chrominance DC quantization factor in version 9e.

3.2 Decompression

We start with default parameters for each version to obtain our baseline result for decompression. By changing different parameters we relate to the baseline and report the sets of versions with equal output. The parameters under investigation are chroma upsampling (simple scaling and fancy upsampling) for a number of factors, and the DCT method. The observed mismatches are reported in Figure 6.

![Figure 5: System model for the decompression experiments](image-url)

**Baseline.** When decompressing with default parameters we see three sets of versions: (1) libjpeg 6b together with libjpeg–turbo, (2) libjpeg 7 to 9, and (3) libjpeg 9a and later. Observe that the sets (2) and (3) differ from those produced by the baseline experiments for compression. No mismatches are observed for grayscale.

**Color Upsampling.** The experiments with color upsampling are carried out for the same subsampling factors as in the compression experiments (Sect. 3.1) for simple and fancy upsampling.

We find that if no subsampling or any kind of simple upsampling is used, the versions (1) 6b and turbo and (2) version 7 to 9 produce identical outputs. If fancy upsampling is selected, then whenever no subsampling factor is a power of two, we see identical results for the described sets (1) and (2). For example, 4:4:4 (no subsampling) and 3:1:1 (no power of two) produces outputs where no distinction between the versions 6b to 9 including turbo can be made. While this pretty much resembles the behavior observed in the compression experiments, we find another difference in decompression: if any of the chroma channels has been subsampled with vertical factor 2 and horizontal 1 (4:4:0), libjpeg 6b and libjpeg–turbo diverge. We can attribute the latter to an optimization introduced in libjpeg–turbo, implemented in the function h1v2_fancy_upsample.

**DCT Method.** Since the DCT method may interact with DCT scaling, we test all three DCT methods for color images with and without chroma subsampling (4:2:0), and grayscale for completeness. Modifying the baseline to the DCT method ifast produces four sets: (1) 6b and turbo, (2) versions 7 to 9, (3) 9a, and (4) 9b to 9e. The DCT method float partitions the sets (1) and (2) and produces a total of 6 sets (see Figure 6). Switching to simple or no upsampling reduces the number of sets for all three DCT methods, however in different ways. A commonality is that version 6b becomes identical to version 7 (see Fig. 6). Unexpectedly, we do observe differences for grayscale images when using the DCT methods fast or float. The DCT method ifast produces two sets: (1) 6b to 9a and (2) 9b to
9e. For the DCT method float we observe unusual sets: (1) 6b with 7 to 8, (2) turbo with 8a to 9a, and (3) 9b to 9e. This suggests that libjpeg–turbo adopted some changes from versions later than 6b. In all other experiments we observe turbo generating either identical results as 6b or forming a singleton.

Quality. To test the influence of the QF on decompression, we compress color and grayscale images with default parameters and vary the QF between 50 and 100. We observe the same pattern as seen for the baseline experiment (QF 75). The compression quality does not seem to impact version differences for decompression.

3.3 Unaligned Input Dimensions
Recall that DCT transforms block of $8 \times 8$ pixels. When image dimensions are not multiples of 8, blocks on the edges are partially outside of the image. Such cases are not standardized and thus seem prone to potential version mismatches.

To generate different edge blocks, we crop the input images to a height and width of various modulo of 8 before compression. For experiments without chroma subsampling we do so by adding the offsets 1, 2, 4, 7 and 8 in both directions to the input dimensions before cropping the images from the center. An $8 \times 8$ block in the chroma component at 4:2:0 subsampling corresponds to a $16 \times 16$ macro-block in the original image. For the experiments with chroma subsampling we therefore add the offsets 1, 2, 3, 7, 8, 9, 15, and 16 before cropping the images. Contrary to our hypothesis, we do not find any new mismatches for compression or decompression.

3.4 Magnitude of Differences
To assess the effect of differences between libjpeg versions on applications in multimedia security, we quantify the differences in the baseline case and compare them to parameter changes for the same version, whose effects on applications are better understood.

Table 2 shows the results for compression. Between all three sets, over 99.3 % of the 1,000 test images differ in at least one DCT coefficient. This is close to the 100 % response to parameter changes within version 9e. To get a better idea of the magnitude of changes within compressed images, we calculate the share of all mismatching quantized DCT coefficients per image and report robust statistics of the distribution over all images, namely quantiles $q_5$, $q_{25}$, and the median (i.e., $q_{50}$). We take logs to focus on the order of magnitude. The fewest DCT coefficients change between version 6b/turbo and 7–9d (about 0.1 % per median), whereas the transition to version 9e affects about four times as many coefficients. Observe that differences between versions are of the same magnitude as switching between DCT methods islow and ifast, which has been shown influential in forensics applications [18]. For perspective, this is much more subtle than changes in quality, which affects between 25 and 75 % of all quantized DCT coefficients for the scenarios shown in Table 2. It requires effort to deal with this in steganalysis [23].

Table 3 shows the results for decompression. Exchanging version 6b/turbo with any other version affects almost every image, whereas the transition from versions 7–9 to 9a–9e leaves 73 % of the images unchanged. This bounds the possibility to detect the libjpeg version used to decompress from any given image. As the output of decompression is in the spatial domain, the peak signal-to-noise ratio (PSNR) is a suitable metric to analyze the distortion caused by version differences. Observe that version 6b/turbo versus any other version has a median PSNR of around 50 dB, comparable to the distortion introduced by high quality JPEG compression. The lower end of the range has values below 40 dB, which can become visible. The differences between versions 7–9 and 9a–9e are tiny, with PSNR above 80 dB. They are so small that rows one and three in the table show identical numbers at the given precision; we confirmed that the underlying data varies. Comparing to parameter changes within version 9e, we note that the version differences can be more pronounced than changes to the DCT method, but are far less influential than the reported changes in quality.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Mismatching images (of 1000)</th>
<th>Share of mismatching DCT coefficients (log10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between versions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6b/turbo vs 7–9d</td>
<td>995</td>
<td>$-3.61$ $-3.04$ $-2.51$</td>
</tr>
<tr>
<td>6b/turbo vs 9e</td>
<td>995</td>
<td>$-2.85$ $-2.34$ $-2.13$</td>
</tr>
<tr>
<td>7–9d vs 9e</td>
<td>993</td>
<td>$-3.00$ $-2.45$ $-2.28$</td>
</tr>
<tr>
<td>Within one version (9e)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCT islow vs fast</td>
<td>1000</td>
<td>$-2.40$ $-2.15$ $-2.01$</td>
</tr>
<tr>
<td>DCT islow vs float</td>
<td>1000</td>
<td>$-3.10$ $-2.80$ $-2.68$</td>
</tr>
<tr>
<td>QF75 vs 90</td>
<td>1000</td>
<td>$-0.99$ $-0.63$ $-0.36$</td>
</tr>
<tr>
<td>QF90 vs 95</td>
<td>1000</td>
<td>$-0.83$ $-0.52$ $-0.28$</td>
</tr>
<tr>
<td>QF95 vs 100</td>
<td>1000</td>
<td>$-0.37$ $-0.14$ $-0.07$</td>
</tr>
</tbody>
</table>
We confirm that JPEG compression and decompression using Matlab’s libjpeg at compile time. The pre-compiled MEX file uses AVX2, and an ARM-based Apple M1 (2020), supporting Neon. The acceleration of libjpeg-turbo is based on vectorization using architecture-dependent SIMD instructions (MMX, SSE2, AVX2, Neon, and Altivec) [1]. This raises the question whether the output of libjpeg-turbo is consistent across architectures supporting different SIMD instruction sets. To get some indication, we run our experiments on an AMD Ryzen 7 4700U, supporting MMX, SSE, and AVX2, and an ARM-based Apple M1 (2020), supporting Neon. The outputs of both architectures are identical. We ensure that libjpeg-turbo actually uses SIMD code by comparing the execution time with version 6b, which is the source of the libjpeg-turbo branch and does not use any SIMD acceleration. Libjpeg-turbo compresses significantly faster than 6b on both tested architectures.

3.5 Libjpeg-turbo on Different Architectures

The acceleration of libjpeg-turbo is based on vectorization using architecture-dependent SIMD instructions (MMX, SSE2, AVX2, Neon, and Altivec) [1]. This raises the question whether the output of libjpeg-turbo is consistent across architectures supporting different SIMD instruction sets. To get some indication, we run our experiments on an AMD Ryzen 7 4700U, supporting MMX, SSE, and AVX2, and an ARM-based Apple M1 (2020), supporting Neon. The outputs of both architectures are identical. We ensure that libjpeg-turbo actually uses SIMD code by comparing the execution time with version 6b, which is the source of the libjpeg-turbo branch and does not use any SIMD acceleration. Libjpeg-turbo compresses significantly faster than 6b on both tested architectures.

3.6 Libjpeg Under the Hood

We confirm that JPEG compression and decompression using Matlab’s imread/imwrite functions produce identical results to those of libjpeg 6b. Salle’s Matlab JPEG Toolbox [27] uses the system-wide libjpeg at compile time. The pre-compiled MEX file uses version 6b. The version does not seem to matter in its main use case of reading DCT coefficients. All versions are identical at this level.

We briefly inspected the compression and decompression functions of the Python packages PIL, matplotlib, and opencv. The first two use libjpeg, but the version varies between systems. The compression results of opencv did not match any of the libraries in our experiments. The compression results of imread/ imwrite functions produce identical results to those of libjpeg 6b. Salle’s Matlab JPEG Toolbox [27] uses the system-wide libjpeg at compile time. The pre-compiled MEX file uses version 6b. The version does not seem to matter in its main use case of reading DCT coefficients. All versions are identical at this level.

4 DISCUSSION

We can draw some lessons from our systematic comparison of numerical differences between libjpeg versions. First, the decompression functions of libjpeg have changed more often between versions than compression functions. Second, the libjpeg version does not seem to matter when working with grayscale images, unless one deviates from the default DCT method (which is rarely reported). This means that many of the “classic” results in the steganalysis and digital forensic literature of the 2000s—the era of libjpeg version 6b—may still be valid. Most of them are (unduly [17]) limited to grayscale. Third, when working with color, the libjpeg version does matter. Under certain parameter constellations, the 14 versions in our experiments produce six different outputs. This hints at a potential vulnerability of security-related applications implemented in forensics or steganalysis and should encourage users and researchers to control and carefully report which JPEG implementation was used: know your library. Future work must establish to what extent the differences found here carry through to forensic detectors, steganalysis, or watermarking.

This work has limitations. First, our results are limited to the parameter and input space covered. One could consider static program analysis to verifiably find all version differences, e.g., using interval arithmetic [19]. Second, we only consider a single version of libjpeg-turbo and omit mozjpeg. Forensics practitioners may encounter all of them and call for an extension of this study to those versions. They might also ask for a method to detect the exact compression library from a given image, refining earlier attempts [4].

Acknowledgment. This work is funded by the EU’s Horizon 2020 programme under grant agreement No. 101021687 (UNCOVER).

REFERENCES


Table 3: Magnitude of changes in decompression

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Mism. images (of 1000)</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>q5</td>
</tr>
<tr>
<td><strong>Between versions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6b/turbo vs 7–9</td>
<td>990</td>
<td>39.61</td>
</tr>
<tr>
<td>7–9 vs 9a–9e</td>
<td>267</td>
<td>81.29</td>
</tr>
<tr>
<td>6b/turbo vs 9a–9e</td>
<td>990</td>
<td>39.61</td>
</tr>
<tr>
<td><strong>Within one version (9e)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCT islow vs ifast</td>
<td>998</td>
<td>45.38</td>
</tr>
<tr>
<td>– &quot; – for grayscale</td>
<td>997</td>
<td>51.25</td>
</tr>
<tr>
<td>DCT islow vs float</td>
<td>997</td>
<td>52.88</td>
</tr>
<tr>
<td>– &quot; – for grayscale</td>
<td>997</td>
<td>57.17</td>
</tr>
<tr>
<td>QF75 vs 90</td>
<td>998</td>
<td>26.60</td>
</tr>
<tr>
<td>QF90 vs 95</td>
<td>998</td>
<td>29.58</td>
</tr>
<tr>
<td>QF95 vs 100</td>
<td>998</td>
<td>35.91</td>
</tr>
</tbody>
</table>


