Progressive JPEGs in the Wild: Implications for Information Hiding and Forensics

Nora Hofer
Universität Innsbruck
Innsbruck, Austria
nora.hofer@uibk.ac.at

Rainer Böhme
Universität Innsbruck
Innsbruck, Austria
rainer.boehme@uibk.ac.at

ABSTRACT

JPEG images stored in progressive mode have become more prevalent recently. An estimated 30% of all JPEG images on the most popular websites use progressive mode. Presumably, this surge is caused by the adoption of *MozJPEG*, an open-source library designed for web publishers. So far, the optimizations used by *MozJPEG* have not been considered by the multimedia security community, although they are highly relevant. The goal of this paper is to document these optimizations and make them accessible to the research community. Most notably, we find that Trellis optimization in *MozJPEG* modifies quantized DCT coefficients in order to improve the rate–distortion tradeoff using a perceptual model based on PSNR-HVS. This may compromise the reliability of known methods in steganography, steganalysis, and image forensics when dealing with images compressed with *MozJPEG*. We also find that the type and order of scans in progressive mode, which *MozJPEG* adjusts to the image, offer novel cues that can aid forensic source identification.

CCS CONCEPTS

• **Computing methodologies → Image compression**; • **Applied computing → Evidence collection, storage and analysis**; • **Security and privacy**;

KEYWORDS

Progressive JPEG, MozJPEG, Trellis quantization, scan script optimization, image forensics

ACM Reference Format:
Nora Hofer and Rainer Böhme. 2023. Progressive JPEGs in the Wild: Implications for Information Hiding and Forensics. In *Proceedings of the 2023 ACM Workshop on Information Hiding and Multimedia Security (IH&MMSec '23), June 28–30, 2023, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3577163.3595097

1 INTRODUCTION

JPEG is a popular standard for the compression and decompression of digital images. Introduced in 1991, it is now supported by countless applications [20] and more than 75% of all websites including digital images use JPEG [37]. JPEG aims at removing imperceptible information, and hence reducing the file size, while preserving the perceptual quality of an image. The JPEG standard [38] defines different modes, including the "baseline" sequential mode and the progressive mode. While the sequential mode encodes images as a whole, the progressive mode partitions image data into several scans, allowing decoders to display a low-quality version of an image even before all image data is received, e. g., via a slow communication link. The image quality is then gradually improved as more scans are received and decoded.

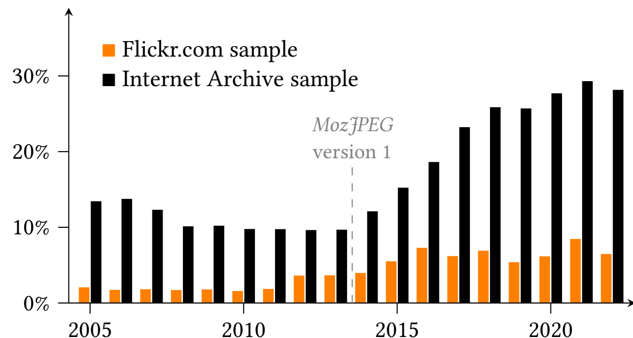
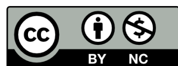


Figure 1: Prevalence of progressive JPEG images on the web. The bars show the share of progressive mode among all sampled JPEGs per year and data source. Note the increase after *MozJPEG* was released.

Although the progressive mode has been part of the standard from the very start, many applications do not use it by default, or not at all. Reports of bugs in browsers when displaying progressive JPEGs have led to recommendations against their use [10]. As a result, the multimedia security research community has barely studied the specifics of progressive JPEGs.

While ignoring the progressive mode was perhaps justifiable in the 1990s and 2000s, when the community was formed, the reality has changed in recent years. Figure 1 reports results of an ad-hoc crawl of roughly 200.000 images from two sources of historical JPEGs on the web: the image sharing platform Flickr.com, used by amateur and professional photographers; and a sample of images from the 2022 Tranco top-5000 websites [26] archived in the Internet Archive's Wayback Machine [32]. While less than 2% of the images on Flickr were progressive in the 2000s, this share more than tripled after 2020. The Internet Archive sample exhibits a similar growth, however from a higher baseline. Today, about one in three JPEG images on the web is progressive.

The increase of progressive JPEGs found after 2014 can likely be explained with the release of *MozJPEG*, an open-source library, which outputs progressive mode by default. Its declared target



This work is licensed under a Creative Commons Attribution-NonCommercial International 4.0 License.

IH&MMSec '23, June 28–30, 2023, Chicago, IL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0054-5/23/06.
https://doi.org/10.1145/3577163.3595097

group are web publishers. Therefore, the library is tuned specifically for compression with the aim to improve end user experience. Shorter loading times of websites with many images and allowing modern browsers to progressively decode both contribute to this objective. Demand for these features go hand in hand with the evolution of web protocols, like SPDY/HTTP2 [17] and QUIC/HTTP3 [5, 22]. Both successors of HTTP enable connection multiplexing, which allows servers to interleave scans of different progressive images and thus better control the rendering of complex websites. As a result, large social networking platforms quickly adopted progressive JPEG, as documented in blog posts by Instagram [14] in 2015, Yelp [1] in 2017, Facebook [2] in 2018, and Twitter [36] in 2023, among others. Moreover, popular messenger services such as WhatsApp and Telegram use progressive JPEG. It is also the default mode for libraries like Fresco [15] that help developers to deal with images on Android.

While exploring the reasons behind the adoption of *MozJPEG* might be interesting, in this work we focus on its consequences. Since many methods in multimedia security rely on subtle traces in the signal originating from compression and decompression operations, it is important to understand optimizations implemented in popular implementations. Researchers tended to assume that their methods generalize to progressive mode, arguing that it merely changes the order of encoding in the JPEG file, but does not modify the signal itself. However, *MozJPEG* invalidates this assumption. In a nutshell: our community should not continue to ignore progressive mode images because they are relevant in practice.

In this paper we address this matter and

- analyze the internals of *MozJPEG*, specifically its default Trellis optimization, which changes DCT coefficients in order to find a rate–distortion tradeoff;
- document characteristic traces in the DCT domain that Trellis optimization leaves behind in output images;
- describe how progressive-mode scan scripts interact with the compression pipeline, and
- discuss the implications for steganalysis, steganography, image forensics, and watermarking.

The remainder of this paper is organized as follows. Section 2 recalls key concepts of JPEG compression, with emphasis on the bit stream encoding and the progressive mode. Section 3 explains how *MozJPEG* differs from commonly known image compression libraries. It details Trellis optimization, the perceptual model, and scan optimization. Section 4 presents the results of our experiments on the effects of *MozJPEG* on image data. Section 5 discusses implications for our research community, before Section 6 concludes.

2 BACKGROUND

We recall JPEG compression with special emphasis on the bit stream encoding as this is relevant for the rate–distortion optimization. We then expand on the progressive mode before reviewing popular implementations.

2.1 JPEG in a Nutshell

An input image in spatial domain representation is converted from the *RGB* to the *YCbCr* color space. This process separates the luminance channel Y from the chrominance channels Cb and Cr. As the human eye is less sensitive to changes in brightness than to changes in color, the chrominance channels can be sub-sampled to increase the compression ratio. Typical 4:2:0 subsampling halves each dimension of the chroma channels, i. e., keeping one quarter of the information. All channels are divided into blocks of 8×8 pixels. Each block is transformed to the frequency domain using the Discrete Cosine Transform (DCT). The resulting coefficients are divided by subband-specific quantization factors before rounding to the nearest integer. The quantization factors are derived from an adjustable quality factor (QF), which is commonly chosen between 75 and 100. Lower QFs imply larger quantization factors, which in turn result in smaller quantized coefficient values and more zeros.

JPEG defines a special source coder which combines Huffman encoding with run-length encoding of zeros. High-frequency coefficients are often quantized to zero, therefore a zigzag arrangement yields longer sequences of consecutive zeros, also called *zero runs*. The DC coefficient is treated separately and not detailed here for

Table 1: Variable-length encoding of coefficient values

Size	Coefficient values		Bit sequence	
0	0		–	
1	**−1**	**1**	0	1
2	**−3**, −2	2, **3**	00, 01	10, 11
3	**−7**,. . . , −4	4,. . . , **7**	000,. . . , 011	100,. . . , 111
4	**−15**,. . . , −8	8,. . . , **15**	0000,. . . , 0111	1000,. . . , 1111
5	**−31**,. . . , −16	16,. . . , **31**	00000,. . . , 01111	10000,. . . , 11111
⋮			. . .	

Table adapted from [38]; candidates in boldface (see Sec. 3.3).

Table 2: Control bytes encoding the size of the coefficient value in bits and the preceding number of zeros (NZ). The smaller numbers in parentheses show the bit length of the control bytes for a specific Huffman table example.

	Bits to store coefficient value								
NZ	**0**	**1**	**2**	**3**	**4**	**5**	**. . .**	**14**	**15**
0	EOB (3)	01 (2)	02 (3)	03 (4)	04 (5)	05 (6)	. . .	14 (0)	15 (0)
1	-	17 (3)	18 (5)	19 (6)	20 (8)	21 (9)	. . .	30 (0)	31 (0)
2	-	33 (5)	34 (7)	35 (10)	36 (11)	37 (12)	. . .	46 (0)	47 (0)
3	-	49 (5)	50 (8)	51 (11)	52 (13)	53 (14)	. . .	62 (0)	63 (0)
4	-	65 (5)	66 (8)	67 (10)	68 (11)	69 (14)	. . .	78 (0)	79 (0)
5	-	81 (6)	82 (9)	83 (12)	84 (13)	85 (14)	. . .	94 (0)	95 (0)
⋮									
14	-	225 (11)	226 (14)	227 (14)	228 (14)	229 (14)	. . .	238 (0)	239 (0)
15	ZRL (11)	241 (13)	242 (14)	243 (14)	244 (14)	245 (14)	. . .	254 (0)	255 (0)

Table adapted from [38].

brevity. Interestingly, the actual values of non-zero coefficients are not subject to Huffman encoding. Instead, the standard defines a variable-length encoding scheme, shown in Table 1. The actual stream is composed of alternating *control bytes* and *variable-length coefficient values*. The control bytes combine the size tag with an optional number of zeros (NZ) preceding the coefficient. Table 2 illustrates the control bytes, also indicating special symbols, such as end-of-block (EOB) and zero run length (ZRL).

Only the control bytes are Huffman-encoded using tables stored in the file. Table 2 is annotated with the number of bits required to store each displayed control byte using an example Huffman table from the Y channel of an image compressed with QF 75. For example, the AC coefficient sequence $(3, 0, 0, 8, 0, 4)$ is encoded as follows:

$$\overbrace{100}^{(0,2)}\;\underbrace{11}_{3},\;\overbrace{111\,1110\,0100}^{(2,4)},\;\underbrace{1000}_{8},\;\overbrace{11\,0111}^{(1,3)}\;\underbrace{100}_{4}\;,$$

where Huffman-encoded control bytes are annotated above and variable-length coefficient values are annotated below the bit stream.

2.2 Progressive Mode

While baseline JPEG uses the sequential mode, the standard also defines a progressive mode [38]. It partitions the information before lossless encoding into several scans. This enables to store all necessary data for a low-quality version of the complete image in the first scan, i. e, at the beginning of the file. Subsequent scans refine the transmitted version of the image until the full image quality is reached. In situations where a JPEG file is transmitted over a slow communication link, a decoder can quickly produce a low-quality image and then gradually improve the displayed quality as more scans are received [21]. After all scans are complete, the final image is identical to that of a sequential JPEG file compressed with the same settings.

The partitioning of image data is specified in the *scan script*. This script can combine *spectral selection*, where lower-frequency subbands are transmitted before higher-frequency subbands, with *successive approximation*, where bits of lower significance are omitted initially and supplied in later scans [25]. Figure 2 illustrates a typical scan script of a grayscale image. The DCT-transformed data is arranged as a cube where the axes represent the subbands (in zig-zag order), the coefficient bits (from MSB to LSB), and the block index. Spectral selection slices the cube in rows, whereas successive approximations cut the cube into columns.

Figure 3 shows the visual effect of progressive decoding of a 768×128 color image compressed with QF 99, default chroma subsampling 4:2:0 and the scan script shown in Figure 4. The image data is partitioned into a total of nine scans using both spectral selection and successive approximation. From left to right, each part of the figure shows an increasing number of scans. We combine corresponding scans for both chroma channels, although they must be stored in separate scans in the file. The initial Scan 1 contains the DC coefficient band without LSB, resulting in an image of 256 blocks representing the block average color. Scan 2 contains the first five low-frequency AC coefficient bands of the luminance channel, excluding the LSB. Scans 3 and 4 contain all AC coefficient

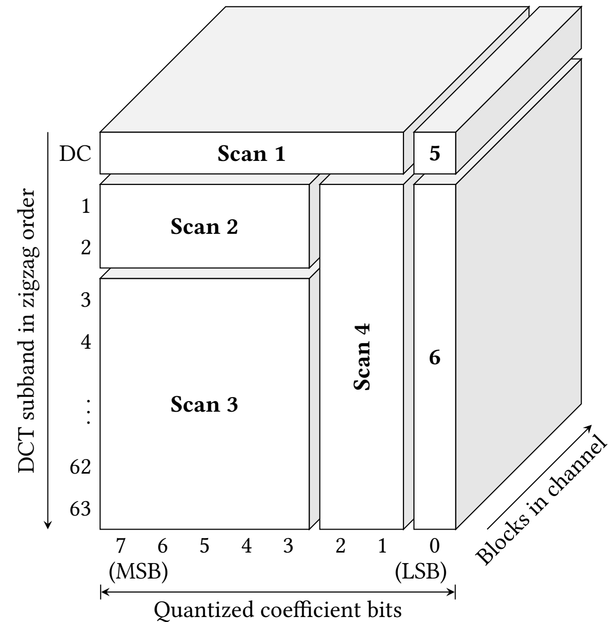


Figure 2: Partitioning of a grayscale image into six scans according to a scan script. Illustration adapted from [38].

bands of both chrominance channels, again excluding the LSB. Scan 5 contains all remaining AC coefficient bands of the luminance channel, again excluding the LSB. At this point, all but the least significant bits of all coefficients are sent. Scan 6 transmits the LSBs of all channels of the DC coefficient bands, and scans 7 to 9 transmit the LSBs of the AC coefficient bands of all channels. Observe from the cumulative shares of DCT coefficients and compressed file size that the information in the first scans is compressed at a lower rate than the refinements in the later scans. The reason for this might be that later scans contain mainly zeros and ones, which can be compressed very efficiently with tailored Huffman tables.

So far, the progressive mode has not been in the center of attention in the research area of multimedia security. We are aware of [35], which proposes selective image encryption specifically for scans in progressive mode. Another innovative use is described in [29]. The authors observe that the set of custom Huffman tables of progressive JPEG images increase the entropy of the file header, allowing to uniquely identify images from header information only.

2.3 Popular Implementations

Many software packages build on the open-source C library *libjpeg*. *libjpeg* has been developed by the Independent JPEG Group [25] since 1991. In 2010, *libjpeg–turbo* was created as a fork of *libjpeg* with the aim of improving the decompression and compression performance by using optimized platform-specific SIMD instructions [28]. Both libraries support the progressive mode, although not as their default.

In 2014, Mozilla forked *libjpeg–turbo* into *MozJPEG* [30] to optimize it for a different objective. *MozJPEG* aims to achieve higher

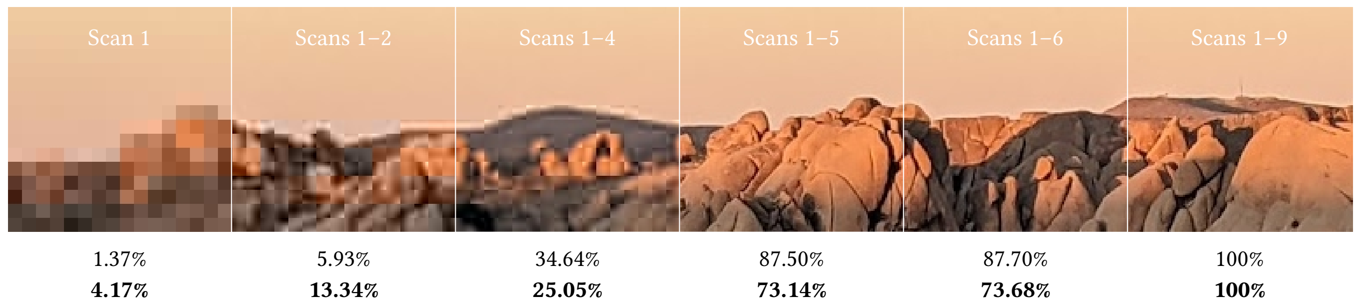


Figure 3: The visual effect of progressive decoding. Percentages show the cumulative share of DCT data (upper row) and compressed file size (lower row) at different steps of decoding.

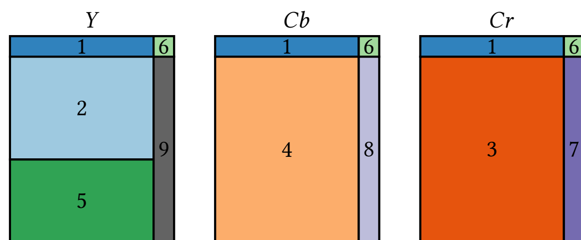


Figure 4: The scan script used to generate Figure 3.

compression rates at the same perceived quality, effectively reducing the loading times of images on the web. To do so, it implements a rate–distortion optimization inspired from trellis quantization [11, 39], it uses progressive mode by default, and selects scan scripts based on the image content.

All three libraries have a common interface and can be used widely interchangeably. The authors of [4] compare different JPEG implementations, including different versions of *libjpeg*, and point out implications for multimedia security. However, due to the different default compression mode, they do not compare to *MozJPEG*. The present work seeks to close this gap.

3 UNDERSTANDING *MOZJPEG*

To the best of our knowledge, the multimedia security community has not devoted much attention to progressive JPEG.[1]

This can be justified by the observation that progressive and sequential modes transmit the very same information on the level of quantized DCT coefficients. Hence, research on steganography, steganalysis, watermarking, and forensics dealing with the signal itself should be unaffected. However, the adoption of *MozJPEG* thwarts this rationale: its Trellis optimization *does* modify the DCT coefficients.

In this section, we explain *MozJPEG*'s modifications to the typical JPEG compression pipeline, thereby commenting on the realized savings for images of varying size and QF. We then explain the perceptual model, the Trellis optimization, and the scan optimization in separate subsections. There are four major versions of *MozJPEG*. The analysis in this paper refers to the latest version 4.1.1 released in August 2022.

[1]An exception is [7] who take into account the changes in DCT coefficients introduced by the rate-distortion optimization of *MozJPEG* while proposing a method for robust steganography.

3.1 *MozJPEG*'s Compression Pipeline

Figure 5 shows a block diagram of *MozJPEG*'s image compression pipeline. The signal path is located in the bottom. Parts where *MozJPEG* innovates compared to other implementations are highlighted in orange. For the gray parts, we refer the reader to the description in Section 2.1. Our convention on the formal notation is summarized in Table 3.

The heart of *MozJPEG* is the rate–distortion optimization. It uses a perceptual model to calculate the distortion implied by reducing non-zero DCT coefficients to values with shorter bit size or even zeroing them out in order to increase the length of zero runs. The distortion is scaled to be comparable to storage bits. The algorithm tries to move each block in each channel independently leftwards in the size-distortion space, illustrated in Figure 5. Small upwards movements are tolerated, as indicated by the indifference line. The estimated size in bits is calculated using a Huffman table specific to the distribution of quantized DCT coefficients in the given image.

Table 3: List of symbols

Symbol	Description
y_i	*un*quantized DCT coefficient value of the i-th subband
y_i^*	quantized DCT coefficient value *before* Trellis
y_i^{**}	quantized DCT coefficient value *after* Trellis
$\boldsymbol{q}$	quantization matrix
q_i	quantization factor of the i-th subband
C	set of suitable candidate values (cf. Table 1)
C_i	set of candidates for a given y_i^*
$c_{i,k}$	elements of C_i
r	run (number of zeros)
n	coefficients in a block, i. e., length of the trellis
$S(y^*, r)$	size (in bits) of a sequence of r zeros followed by the non-zero coefficient y^* after Huffman encoding
$D_i(y^*, y)$	(additive) distortion when the quantized coefficient y^* represents the unquantized value y in subband i
λ	rate–distortion parameter (depends on QF)
κ	cost (sum of size in bits and distortion)

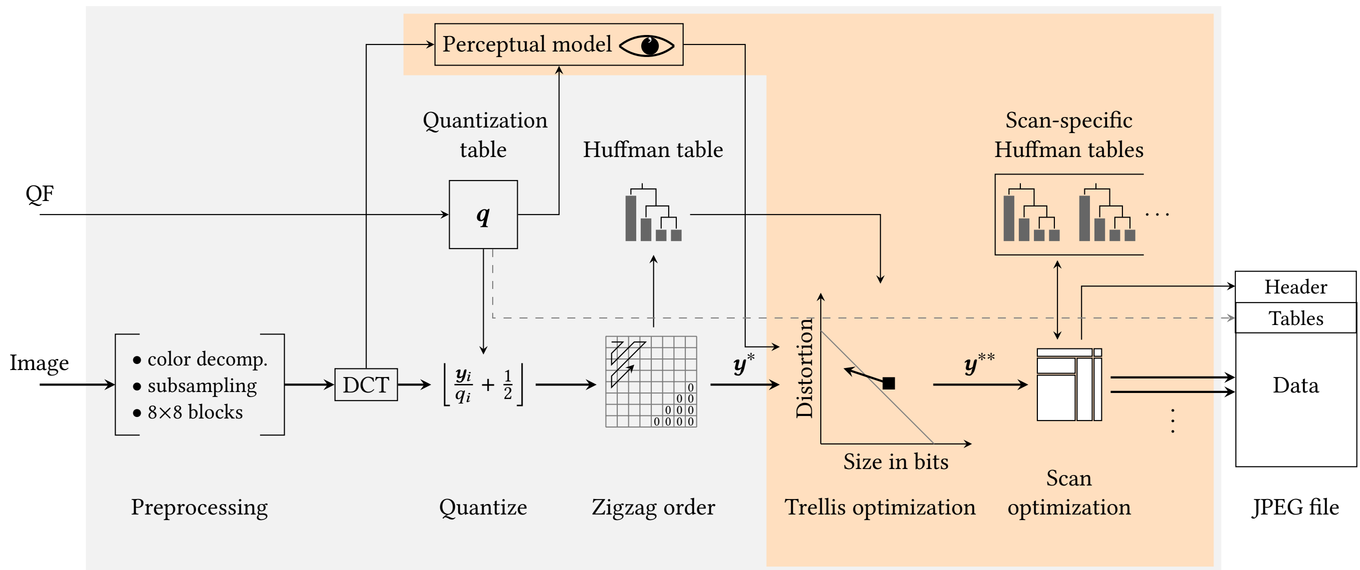


Figure 5: Compression pipeline for progressive JPEG. The orange parts are specific to default *MozJPEG*.

Note that the Trellis optimization is not performed on a scan level but under the assumption of sequential mode. Only the resulting quantized DCT coefficients are passed to the scan optimization. This step jointly optimizes a scan script and a set of corresponding Huffman tables, which go into the output file. If both optimizations are enabled (as by default), the estimated sizes used for Trellis optimization do not necessarily correspond to the actual sizes. This is because the subsequent scan optimization almost certainly generates Huffman tables that encode control bytes with fewer bits. This suggests that repeated optimization passes may exhibit similar convergence behavior as reported for repeated quantization [8, 24].

3.2 Perceptual Model

The purpose of the perceptual model is to estimate the perceived distortion for alternative (i. e., smaller) coefficient values. *MozJPEG* implements the variant of PSNR-HVS described in [13]. This metric, denoted $D_i(y^*, y)$, is based on the square error between the unquantized coefficient y_i and the dequantized coefficient $q_i \cdot y_i^*$, where q_i is the quantization factor in subband i (cf. Table 3). The implied assumption is that the quantization table is a good approximation of the human sensitivity to changes in subbands [13].

An advantage of this model is that the distortion is additive in the DCT domain, unlike many earlier models of the human visual system [31]. This property considerably simplifies the search for a rate–distortion tradeoff as calculating the distortion does not involve transformations between domains and the coefficients can be selected independently with regards to distortion.

3.3 Trellis Optimization

Trellis optimization in general solves the rate–distortion problem by finding the path through a trellis structure that minimizes a cost function [39]. The Viterbi algorithm can find a solution with modest complexity even if the size in bits is variable and non-monotonic in the coefficient value, and the distortion is non-additive across subbands. This way, a coefficient could be rounded upwards with small impact on the size in order to compensate a larger downward rounding of another coefficient, leading to a net reduction in size and a better rate–distortion tradeoff. This general case has been proposed for video codecs [39]. *MozJPEG* takes a much simpler approach.

Specifically, *MozJPEG* exploits two properties:

(1) the additive distortion model (cf. Section 3.2), and
(2) the fact that the variable-size encoding of coefficient values is fixed in the JPEG standard, independent of other coefficients, and monotonic in the absolute value of the coefficient (cf. Table 1).

Since distortion and size are independent for non-zero coefficients, the cost κ is independent, too. The only remaining dependency between AC coefficients in a block arises in the case of zero runs.

Visualizing this observation in a trellis diagram, Figure 6 shows how the number of paths to consider is limited in *MozJPEG*. Observe that each candidate node in Figure 6b has exactly one incoming edge (in orange) from the previous non-zero coefficient, regardless of its value; plus one for each potential run of preceding zeros, implying a quadratic upper bound of the search space. By contrast, the general case has one incoming edge for each possible value of the preceding non-zero coefficient, implying an exponential search space in n.

We now describe *MozJPEG*'s Trellis optimization of zigzag-ordered AC coefficients in a block. The sequence of the difference-encoded DC coefficients of all blocks is optimized similarly, but we omit them here for brevity. Trellis optimization consists of two steps: first it evaluates potential alternative coefficient values with smaller bit sizes and then tries to increase the number and length of

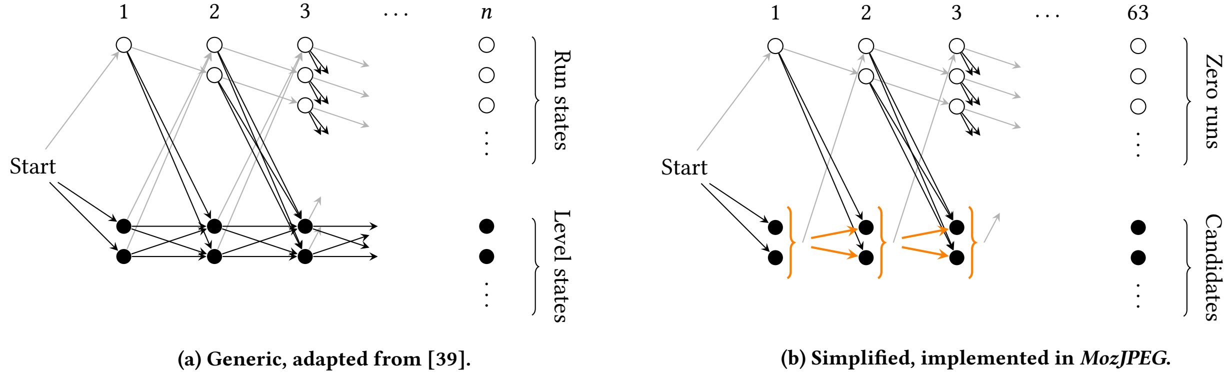


Figure 6: Comparing *MozJPEG*'s Trellis optimization (right) to the general case (left). In *MozJPEG*, the cost function is additive for non-zero AC DCT coefficients. This curbs the path explosion by limiting the path dependencies to zero runs.

zero runs. We denote y_i^* and y_i^{**} as the quantized coefficients *before* and *after* Trellis optimization, respectively. *MozJPEG* constructs a set of possible alternative candidates

$$c_{i,j} \in C_i = \{\forall c \in C : |c| < |y_i^*|\} \cup \{y_i^*\}, \quad (1)$$

where set $C = \{\pm(2^i - 1); i = 1, \ldots, 15\}$ contains all numbers with maximum absolute value per size in bits according to the variable-length encoding of coefficient values (cf. Table 1). These numbers are suitable candidates as they allow to encode a larger original coefficient value with fewer bits while minimizing the distortion.

The size of non-zero coefficients also depends on the number of preceding zeros. *MozJPEG* iterates over all paths with zero runs of varying length r preceding the non-zero coefficient. This causes the quadratic complexity as visible in Fig. 6b. The cost κ is given by

$$\kappa_{i,k,r} = \underbrace{S(c_{i,k}, r)}_{\text{total size in bits}} + \underbrace{D_i(c_{i,k}, y_i)}_{\text{distortion in subband } i} + \underbrace{\sum_{j=i-r}^{i-1} D_j(0, y_j)}_{\text{distortion of } r \text{ zeros}} . \quad (2)$$

Function S returns the size (in bits) of the Huffman-encoded control byte for a sequence of r zeros plus the size of the variable-length coefficient value. Note that more than one control byte may be necessary to encode zero runs larger than 15 (cf. Table 2). Function D is the distortion model as defined in Section 3.2.

The final quantized coefficient after Trellis optimization y_i^{**} is set to $c_{i,k}$ with the lowest $\kappa_{i,k,r}$. If $r > 0$, some non-zero coefficients $y_j^{**}, j < i$ may be set to zero to realize the run.

Example. Table 4 shows a numerical example. The first four unquantized AC DCT coefficients y_i are quantized to y_i^* with quantization factors q_i from *MozJPEG*'s quantization table for QF 75. The table shows the path exploration for $i = 4$ (in bold). The set of candidates C_i includes the original value 8, which has a variable bit size of 4, and all positive numbers with the highest absolute value for each smaller variable bit size, i. e., 7, 3, and 1. The table has one section for the exploration of the paths associated with each candidate. For the original value $c_{4,1} = 8$, the distortion is determined by the quantization error. In our example, the distortion of candidate $c_{4,2} = 7$ is the same, because the quantization error $\|8 \cdot 72 - 540\| = 36$ is exactly half of the quantization factor q_4. Hence, the direction of rounding does not matter in this case. Since 7 can be encoded in two fewer bits — one from the variable-size encoding and the other one from the Huffman table of the control byte — the resulting cost of 35.89 is exactly two "bits" smaller than for the original value. In this example, this costs is the minimum of all rows, hence $y_4^{**} = 7$ will be the result of this optimization step. Note that in order to get the minimum costs, the algorithm needs to calculate them of all other rows shown, which involves the exploration of three additional paths per candidate for potential zero runs. For this purpose, the size column shows the accumulated size of all coefficients up to i. The costs of the example coefficients are prohibitively high. This is different for higher frequency AC coefficients, which tend to be of smaller absolute value. We have chosen $i = 4$ to fit the table in a column.

3.4 Scan Optimization

MozJPEG enables scan optimization by default. It reimplements an idea proposed in the form of a code snippet[2] published by Loren Merritt in 2009. Unlike Trellis optimization, scan optimization does not alter the signal. The algorithm searches 23 variations of scans for the luminance channel and 41 for the chrominance channels to compose the scan script with the shortest output size. This involves deriving the optimal Huffman table for each scan. Our experiments with 100 test images from the Alaska dataset [9] resulted in 86 distinct scan scripts for QF 100, 69 for QF 99, and 35 for QF 75. Observe that the user-experience during progressive decoding over slow connections is not an optimization criterion. We conjecture that large websites therefore supply their own scan scripts.

4 EFFECTS OF MOZJPEG

This section analyzes the impact of *MozJPEG* on file sizes and DCT coefficients experimentally. We use a random sample of 100 never-compressed color images from the Alaska dataset [9]. The images were acquired with different cameras and cropped to 512×512 pixels by the publishers of the dataset.

[2] https://github.com/bsandrow/utils/blob/master/jpegrescan

Table 4: Example of a Trellis iteration for a coefficient with three candidates and three preceding non-zero AC coefficients. The smaller numbers in parentheses show the bit lengths of the Huffman-encoded control byte plus the size of the variable-length coefficient value.

	AC coefficients (zigzag order →)				Size	Distortion	Cost
i	1	2	3	4			
y_i	−574	−635	−107	540			
q_i	64	64	64	72			
y_i^*	−9	−10	−2	**8**			
Candidate $c_{4,1}$ = 8:							
	−9 (5+4)	−10 (5+4)	−2 (3+2)	**8** (5+4)	32	5.89	37.89
	−9 (5+4)	−10 (5+4)	0	**8** (8+4)	30	49.20	79.20
	−9 (5+4)	0	0	**8** (11+4)	24	1639.87	1663.87
	0	0	0	**8** (13+4)	17	2939.60	2956.60
Candidate $c_{4,2}$ = 7:							
	−9 (5+4)	−10 (5+4)	−2 (3+2)	**7** (4+3)	30	5.89	35.89
	−9 (5+4)	−10 (5+4)	0	**7** (6+3)	27	49.33	76.33
	−9 (5+4)	0	0	**7** (10+3)	22	1639.88	1661.88
	0	0	0	**7** (11+3)	14	2939.60	2953.60
Candidate $c_{4,3}$ = 3:							
	−9 (5+4)	−10 (5+4)	−2 (3+2)	**3** (3+2)	28	329.06	357.06
	−9 (5+4)	−10 (5+4)	0	**3** (5+2)	25	372.49	397.49
	−9 (5+4)	0	0	**3** (7+2)	18	1963.04	1981.04
	0	0	0	**3** (8+2)	10	3262.77	3272.77
Candidate $c_{4,4}$ = 1:							
	−9 (5+4)	−10 (5+4)	−2 (3+2)	**1** (2+1)	26	684.53	710.53
	−9 (5+4)	−10 (5+4)	0	**1** (3+1)	22	727.97	749.97
	−9 (5+4)	0	0	**1** (5+1)	15	2318.52	2333.52
	0	0	0	**1** (5+1)	6	3618.24	3624.24

4.1 Effect on File Size

Figure 7 shows how different parts of the *MozJPEG* compression pipeline influence the output size. Solid lines are averages over 100 images with 512×512 pixels; dashed lines represent images downscaled to 48×48, the typical size of icons on social media websites. Both sets of images are compressed with the typical 4:2:0 chroma subsampling using three different QFs, 100, 99, and 75. *MozJPEG*'s default setting with progressive mode, Trellis optimization, and scan optimization is shown in the rightmost column. All numbers are scaled to *MozJPEG*'s sequential mode without any optimization as 100%. For comparison, the sequential default in baseline *libjpeg* version 6b is given on the left.

The difference between *libjpeg* and *MozJPEG* sequential is due to *MozJPEG*'s more aggressive quantization matrices and the use of custom Huffman tables.[3] Our measurements for 512×512 are broadly in line with numbers reported by [27]. This source compares compression ratio and performance of default *MozJPEG* to *libjpeg–turbo*. It finds that *MozJPEG* outputs 20% smaller files while taking on average 25% more time to compress. Note that the study compares *MozJPEG* at version 2.1, which does not apply Trellis optimization to DC coefficients. Icon-sized images seem to benefit significantly more from *MozJPEG*, speculatively a reason for its adoption by large websites.

Turning to the compression options of *MozJPEG*, Trellis optimization offers the largest space savings, followed by scan optimization. The use of the progressive mode alone is advantageous for larger images; only in combination with Trellis and scan optimization it does not blow up the file size of icon-sized images. This suggests that *MozJPEG*'s defaults are chosen very reasonably for the variety of images served on the web.

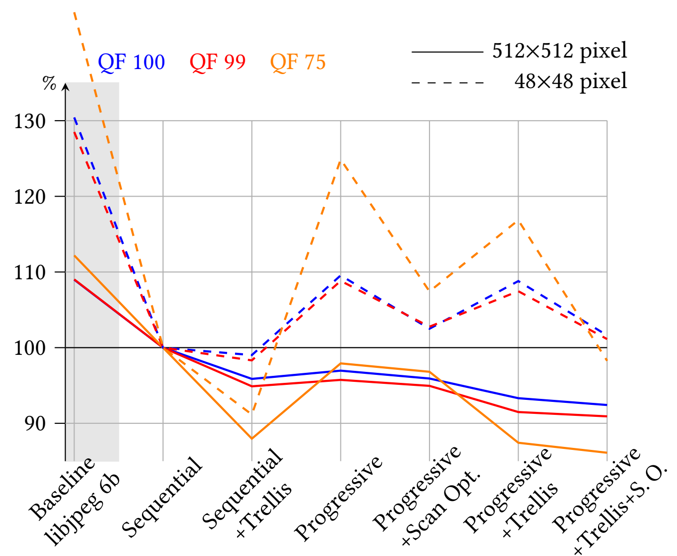


Figure 7: Effect of compression options on file size. Average over 100 color images. Sequential *MozJPEG* with all optimizations disabled = 100%.

4.2 Effect on DCT Coefficients

We measure the effect of Trellis optimization in the frequency domain by comparing quantized AC coefficients of the luminance channel before and after Trellis optimization.

The distribution of DCT coefficients from natural images usually follows a Laplace distribution [34]. Trellis optimization shifts

[3] *libjpeg* and *libjpeg–turbo* use the general-purpose Huffman tables defined in the JPEG standard, which are clearly suboptimal for certain scans in progressive mode.

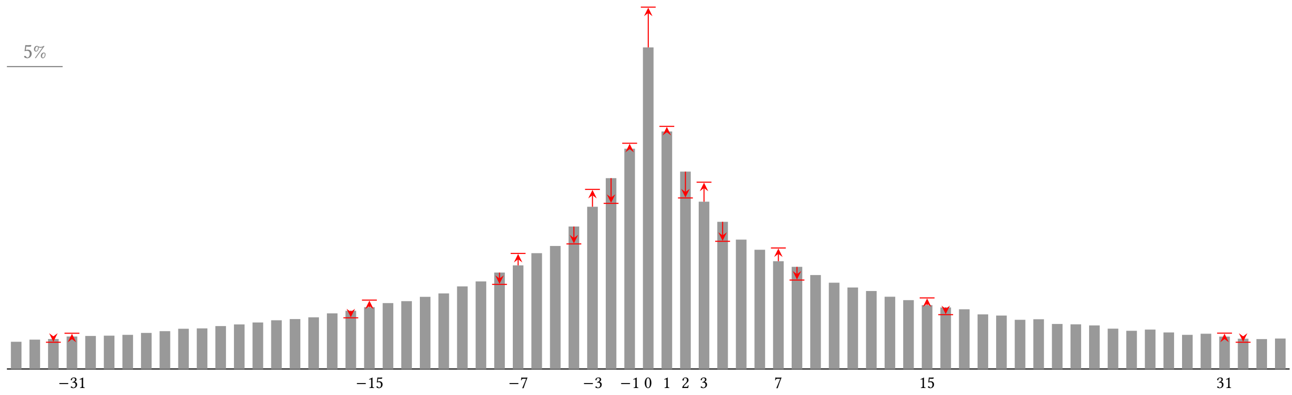


Figure 8: Histograms of the first DCT AC coefficient before (bar) and after (arrow) Trellis optimization for QF 99.

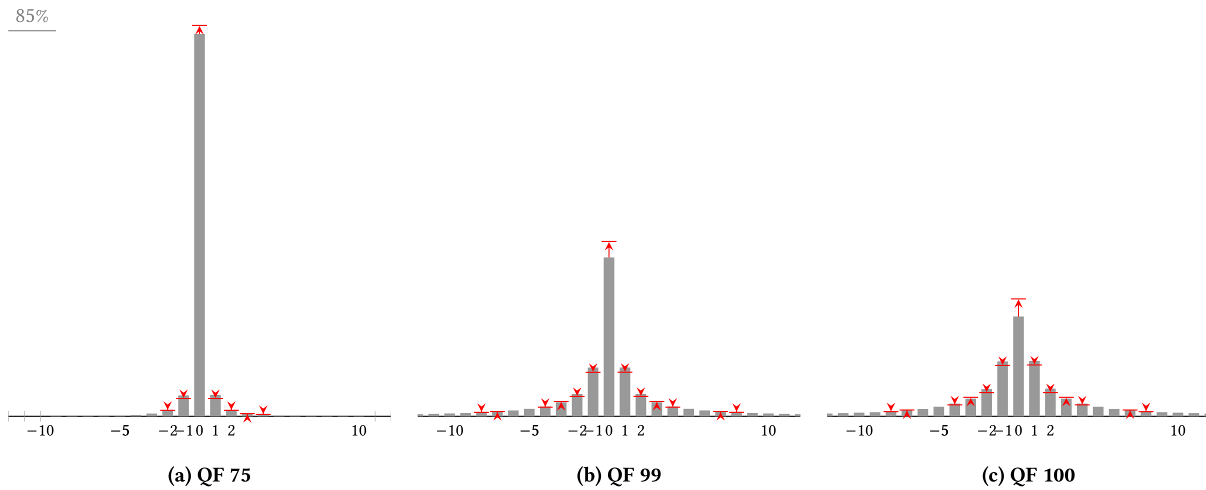


Figure 9: Histograms of all quantized AC DCT coefficients before (bar) and after (arrow) Trellis optimization.

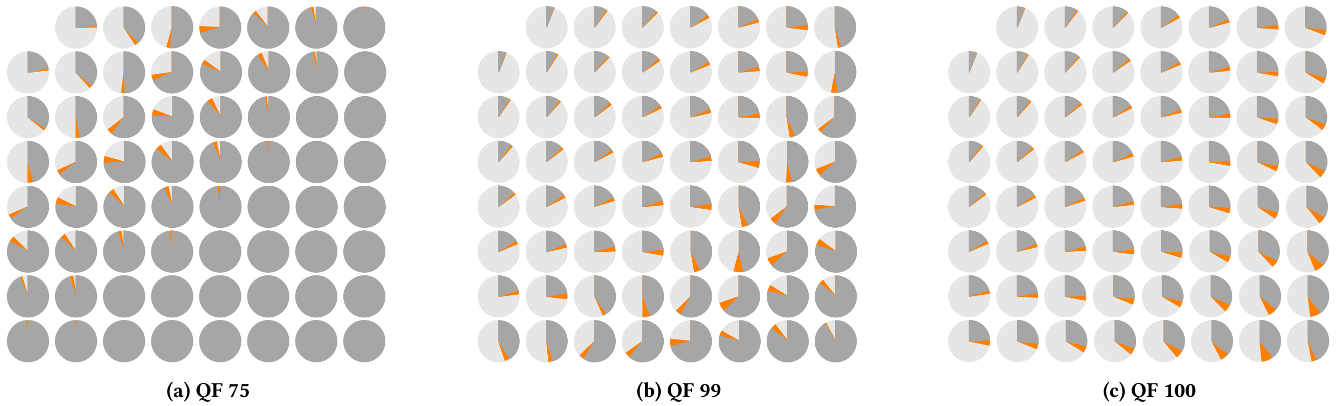
probability mass towards zero, and towards suitable candidate values of shorter variable-length encoding (cf. Table 1). Therefore, we expect characteristic deviations from this discretized distribution for images compressed with *MozJPEG*.

Figure 8 illustrates these deviations for coefficients of the first AC subband of 100 sample images. It is visible that candidate coefficients occur more frequently, while upper neighboring coefficients appear less often. Also, the frequency of zeros increases. Figure 9 shows the same effect for all AC coefficient subbands for different QFs. The same effect is present, albeit less visible for non-zero values due to the increasing number of coefficients equal and close to zero in high-frequency subbands.

Figure 10 shows the share of coefficients being changed to zero broken down by DCT subband and QF. The highest increases in zeros are observed in subbands that already have a high share of zeros. This is plausible as preexisting zero runs can be combined or enlarged with modest additional distortion. Future detectors of *MozJPEG* should weigh the subbands used for the decision by this statistic.

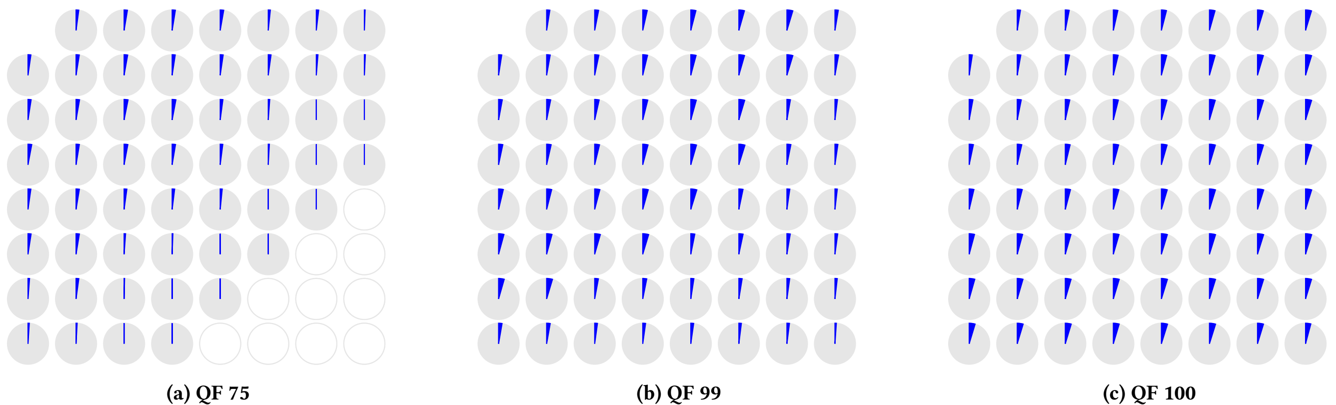
Non-zero coefficients are changed at a small, but relatively consistent rate in all subbands except the ones with a very high share of zeros, as shown in Figure 11. For QF 75, some high-frequency subbands are all zeros; here the ratio is not defined. Figure 12 extends the analysis and shows that the findings hold true for all QFs from 50 to 100. It shows the average share of introduced changes of all AC subbands as well as the first (1,2), and the two next-to-last AC subbands in the purely horizontal (1,7), and vertical (7,1) frequency dimension.

Finally, Figure 13 populates the rate–distortion tradeoff sketched in Figure 5 with empirical data collected from an instrumented version of *MozJPEG*. Each arrow shows how one of 128 randomly selected blocks from a sample image moves in the size–distortion space. The same blocks are selected for each of the three QFs. As expected, the algorithm moves blocks to the upper left, i.e., reducing the size in bits and slightly increasing the distortion, crossing



(a) QF 75 (b) QF 99 (c) QF 100

Figure 10: Share of zeros before (dark) and after (dark + orange) Trellis optimization by DCT subband and JPEG quality factor.



(a) QF 75 (b) QF 99 (c) QF 100

Figure 11: Share of non-zero DCT coefficients modified by Trellis optimization.

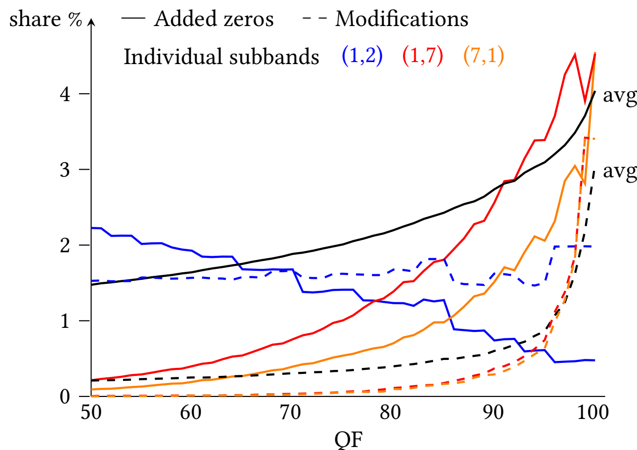


Figure 12: The average share of *additional zeros* and *non-zero modifications* caused by Trellis optimization in the luminance channel of 100 512×512 images from the Alaska dataset for all QFs from 50 to 100.

imaginary indifference lines at 45°. Horizontal left arrows indicate blocks that can be encoded in fewer bits without increasing distortion. This can only happen if the initially quantized coefficient is rounded upwards and the quantization error is exactly half of the quantization factor. Our experiments show that the special case in the numerical example of Section 3.3 actually happens in practice. The concentration of blocks at size 3 for QF 75 can be explained by the bits required to encode the EOB symbol according to the image's Huffman table (cf. Tab. 2).

5 DISCUSSION

The adoption of *MozJPEG* has implications for multimedia forensics in research and practice. In this section we map out avenues for future research.

5.1 Implications for Steganography

Besides one exception [7], all research in JPEG steganography and steganalysis assumes baseline JPEG. All practical steganographic tools we are aware of support sequential mode only. Consequently, a progressive JPEG most likely does not contain steganography embedded with the known tools.

However, steganalysts may not only benefit.

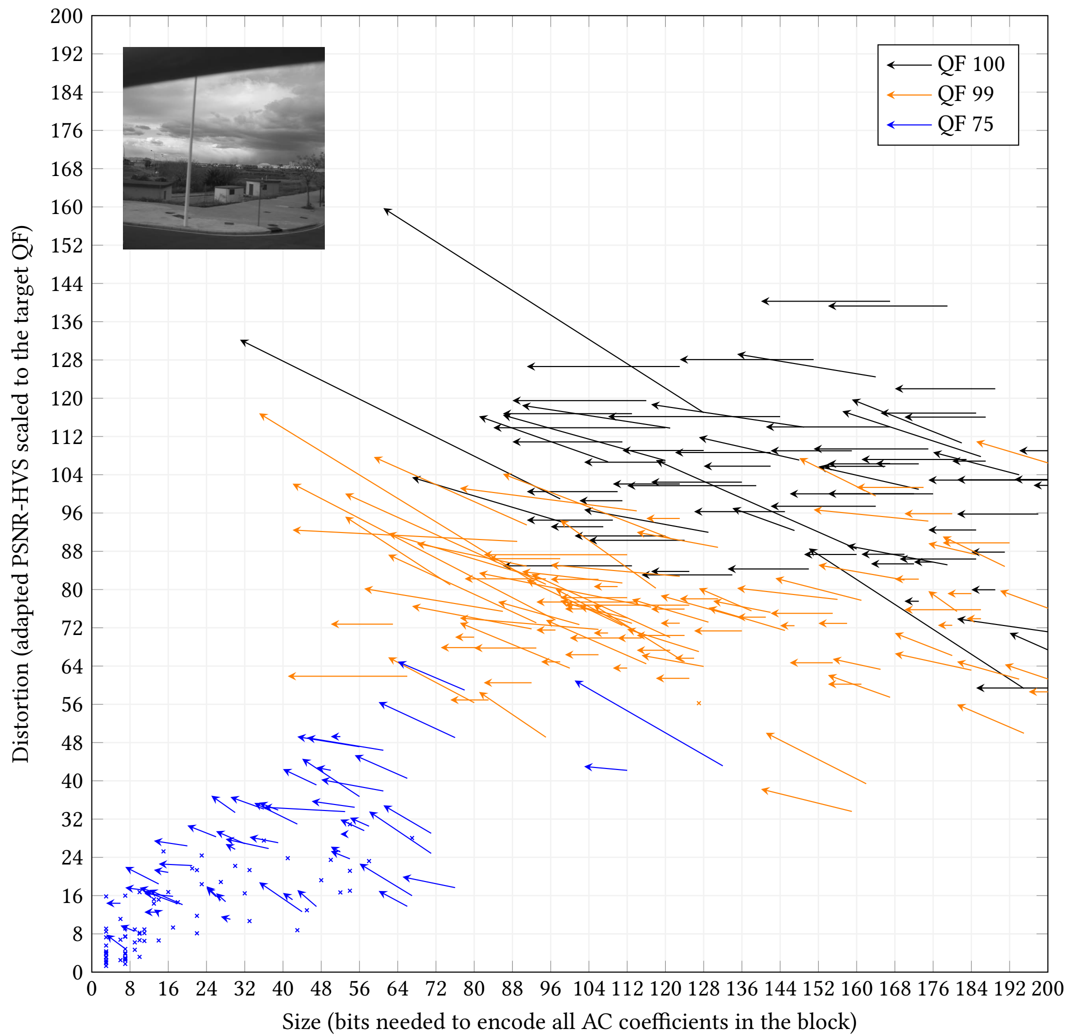


Figure 13: The effect of *MozJPEG*'s Trellis optimization on selected blocks in the luminance channel for an example image from the Alaska dataset [9].

Modern learning-based steganalysis is sensitive to cover–source mismatch: detectors degrade significantly if they are run on material that deviates from the training data [18]. Early evidence suggests that the choice of the JPEG compression library may contribute to this mismatch [3]. Since *MozJPEG* is being widely adopted, but barely considered by researchers, the true extent of this error remains unknown. Specifically, the changes to the DCT coefficients introduced by Trellis optimization resemble in part the changes of popular embedding functions. For example, F5 [40] decrements the absolute value of DCT coefficients and inflates the number of zeros, quite akin *MozJPEG*. Consequently, pristine images compressed with *MozJPEG*'s default may appear as false positives in steganalysis. More research is required to evaluate and quantify this effect. Also, steganalysis based on JPEG compatibility [12, 16] is sensitive to the very details of the implementation and known methods should be revisited in the light of *MozJPEG*.

Finaly, steganographers might try to mimic these compression artifacts in order to hide secret messages. While the capacity is probably very low given the low number of changes made during Trellis optimization (cf. Figures 10 and 11), such an embedding function could be very secure. Another insight for steganography concerns the generalization of known embedding functions from grayscale (single-channel) to color. An open question in the field is whether independent embedding in all color channels is secure. Since *MozJPEG* optimizes DCT coefficients in each channel independently, there exists at least one benign process which does exactly this. Hence, steganalysis exploiting dependencies in color channels may be less promising than researchers suggest [23].

5.2 Implications for Forensics

Similarly to steganalysis, most JPEG forensic techniques were designed for sequential images. Therefore, these techniques may be

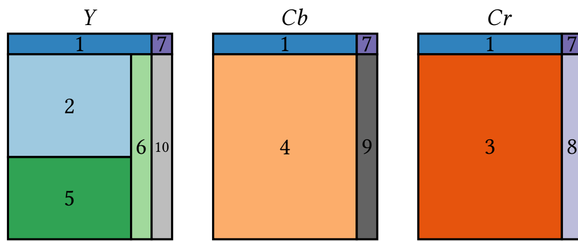


Figure 14: The scan script found in images in the Forchheim database assigned to *Telegram*, *Twitter*, and *Facebook*. The script is identical to the standard scan script used by *libjpeg* and *libjpeg–turbo*.

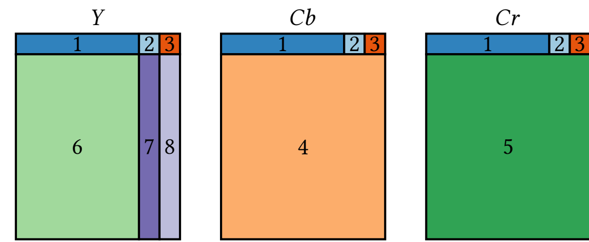


Figure 15: The scan script found in all images in the Forchheim database assigned to *WhatsApp*.

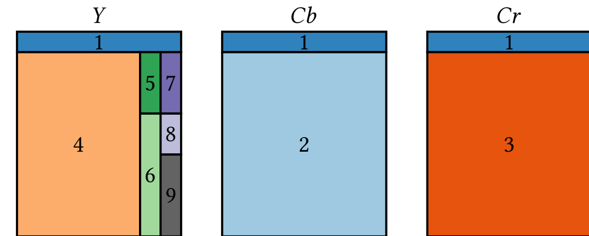


Figure 16: The scan script found in all images in the Forchheim database assigned to *Instagram*.

prone to decision errors if presented with images from *MozJPEG*. For example, techniques for detecting JPEG double compression usually rely on assumptions about the distribution of DCT coefficients. Future research should re-evaluate and, if necessary, adapt existing methods and tools. First and foremost, techniques that rely on assumptions about the distribution of DCT coefficients or repeated quantization [33] seem most affected.

However, *MozJPEG* also provides a number of opportunities for forensic analysis. A common task in forensics is to identify the compression history [6]. A next logical step would be to develop and evaluate a detector for *MozJPEG*.

The very fact that progressive mode has been used, the scan script, and the use of custom Huffman tables may reveal information about the origin and authenticity of an image. Since baseline *libjpeg* and *libjpeg–turbo* use fixed Huffman tables by default, and a fixed scan script if instructed to produce progressive output, there is little identifying information in these entries. As demonstrated in [29], the adoption of *MozJPEG* has changed this.

Another interesting trace is the specific scan script used in progressive images. In a preliminary experiment, we analyzed the scan scripts in social media images from the Forchheim image database [19] and find that 97% use progressive mode. While images assigned to Facebook, Telegram, and Twitter use the default scan script of *libjpeg* and *libjpeg–turbo*, images assigned to WhatsApp and Instagram contain different, distinct scan scripts. These findings suggest that different platforms fine-tune the scan script to their needs. Figures 14–16 show the scan scripts found. More research is needed to validate these findings independently and over time, while controlling the device type and software version of sender and recipient.

5.3 Implications for Watermarking

Custom scan scripts may also have applications in digital watermarking. Compressing a JPEG with a unique custom scan script can serve as a fragile watermark to recognize marked images or to detect recompression if a supposedly marked image does not have the specific script anymore. To reduce the risk that other images accidentally share the marking scan script, it can be designed in a "useless" way, e. g., transmitting less relevant information first.

6 CONCLUSION

Progressive JPEG has become more prevalent than commonly assumed, presumably due to the adoption of *MozJPEG*, an open-source library optimized for web publishers. To the best of our knowledge, we are the first to document the optimizations implemented in this library to make them accessible to the multimedia security community. Our experiments reveal characteristic traces in images compressed with *MozJPEG*. We discuss how these may affect established methods in steganography, steganalysis, image forensics, and watermarking. In particular researchers proposing learning-based methods should in the future include images compressed with *MozJPEG* in their evaluation protocol.

ACKNOWLEDGMENTS

We thank Maximilian Hils for his support on the web crawling measurement and Benedikt Lorch and Martin Beneš for valuable comments on the draft. This work is funded by the EU's Horizon 2020 program under grant agreement No. 101021687 (UNCOVER).

REFERENCES

[1] Stephen Arthur. 2017. Making photos smaller without quality loss. https://engineeringblog.yelp.com/2017/06/making-photos-smaller.html (accessed: Jan 9, 2023).
[2] Tomer Bar. 2018. Faster photos in facebook for IOS. https://engineering.fb.com/2015/01/28/ios/faster-photos-in-facebook-for-ios/ (accessed: Jan 9, 2023).
[3] Martin Beneš, Nora Hofer, and Rainer Böhme. 2022. The effect of the JPEG implementation on the cover-source mismatch error in image steganalysis. In *European Signal Processing Conference*. IEEE, 1057–1061.
[4] Martin Beneš, Nora Hofer, and Rainer Böhme. 2022. Know your library: How the libjpeg version influences compression and decompression results. In *Workshop on Information Hiding and Multimedia Security*. ACM, 19–25.
[5] Mike Bishop et al. 2021. Hypertext transfer protocol version 3 (HTTP/3). *Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-34* (2021).
[6] Jan Butora and Patrick Bas. 2022. High quality JPEG compressor detection via decompression error. In *GRETSI*.

[7] Jan Butora, Pauline Puteaux, and Patrick Bas. 2022. Errorless robust JPEG steganography using outputs of JPEG coders. *arXiv preprint arXiv:2211.04750* (2022).
[8] Matthias Carnein, Pascal Schöttle, and Rainer Böhme. 2015. Forensics of high-quality JPEG images with color subsampling. In *Workshop on Information Forensics and Security*. IEEE, 1–6.
[9] Rémi Cogranne, Quentin Giboulot, and Patrick Bas. 2019. The ALASKA steganalysis challenge: A first step towards steganalysis. In *Workshop on Information Hiding and Multimedia Security*. ACM, 125–137.
[10] Wikimedia commons. 2017. Help:JPEG. https://commons.wikimedia.org/wiki/Help:JPEG, (accessed: Feb 13, 2023).
[11] Matt Crouse and Kannan Ramchandran. 1997. Joint thresholding and quantizer selection for transform image coding: entropy-constrained analysis and applications to baseline JPEG. *Transactions on Image Processing* 6, 2 (1997), 285–297.
[12] Eli Dworetzky and Jessica Fridrich. 2021. JPEG compatibility attack revisited. *Transactions on Information Forensics and Security* (2021).
[13] Karen Egiazarian, Jaakko Astola, Nikolay Ponomarenko, Vladimir Lukin, Federica Battisti, and Marco Carli. 2006. New full-reference quality metrics based on HVS. In *International Workshop on Video Processing and Quality Metrics*, Vol. 4.
[14] Instagram Engineering. 2015. Under the Hood: Instagram in 2015. https://instagram-engineering.com/under-the-hood-instagram-in-2015-8e8aff5ab7c2, (accessed: Feb 13, 2023).
[15] Fresco. 2023. An image management library. https://frescolib.org/, (accessed: Feb 13, 2023).
[16] Jessica Fridrich, Miroslav Goljan, and Rui Du. 2001. Steganalysis based on JPEG compatibility. In *Multimedia Systems and Applications*, Vol. 4518. 275–280.
[17] Andrew Galloni and Kornel Lesiński. 2020. Progressive image streaming. https://blog.cloudflare.com/parallel-streaming-of-progressive-images/, (accessed: Mar 05, 2023).
[18] Quentin Giboulot, Rémi Cogranne, Dirk Borghys, and Patrick Bas. 2020. Effects and solutions of cover-source mismatch in image steganalysis. *Signal Processing: Image Communication* 86 (2020), 115888.
[19] Benjamin Hadwiger and Christian Riess. 2021. The Forchheim image database for camera identification in the wild. In *Pattern Recognition, Computer Vision, and Image Processing*. Springer, 500–515.
[20] Graham Hudson, Alain Léger, Birger Niss, István Sebestyén, and Jørgen Vaaben. 2018. JPEG-1 standard 25 years: past, present, and future reasons for a success. *Journal of Electronic Imaging* 27, 4 (2018), 040901–040901.
[21] Jaehan In, Shahram Shirani, and Faouzi Kossentini. 1998. JPEG compliant efficient progressive image coding. 5 (1998), 2633–2636.
[22] Jana Iyengar, Martin Thomson, et al. 2021. QUIC: A UDP-based multiplexed and secure transport. In *RFC 9000*.
[23] Matthias Kirchner and Rainer Böhme. 2014. "Steganalysis in Technicolor" Boosting WS detection of stego images from CFA-interpolated covers. In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 3982–3986.
[24] ShiYue Lai and Rainer Böhme. 2013. Block convergence in repeated transform coding. In *International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 3028–3032.
[25] Thomas Lane. 1994. Using the IJG JPEG library. https://www.freedesktop.org/wiki/Software/libjpeg/, (accessed: Jan 9, 2023).
[26] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Annual Network and Distributed System Security Symposium*. //tranco-list.eu/list/5Y5GN, (accessed: Mar 05, 2023).
[27] libjpeg turbo. 2017. What About mozjpeg? https://www.libjpeg-turbo.org/About/Mozjpeg, (accessed: Feb 13, 2023).
[28] The libjpeg-turbo Project. 2022. libjpeg–turbo. https://libjpeg-turbo.org/ (accessed: Jan 28, 2023).
[29] Sean McKeown, Gordon Russell, and Petra Leimich. 2018. Fingerprinting JPEGs with optimised Huffman tables. *The Journal of Digital Forensics, Security and Law* 13, 2 (2018).
[30] Mozilla Foundation. 2014. MozJPEG: Improved JPEG encoder. https://github.com/mozilla/mozjpeg (accessed: Jan 28, 2023).
[31] Norman Nill. 1985. A visual model weighted cosine transform for image compression and quality assessment. *Transactions on Communications* 33, 6 (1985), 551–557.
[32] Greg Notess. 2002. The wayback machine: The web's archive. 26, 2 (2002), 59–61.
[33] Cecilia Pasquini and Rainer Bohme. 2018. Towards a theory of JPEG block convergence. In *International Conference on Image Processing*. IEEE, 550–554.
[34] Randall Reininger and Jerry Gibson. 1983. Distributions of the two-dimensional DCT coefficients for images. *Transactions on Communications* 31, 6 (1983), 835–839.
[35] Thomas Stiitz and Andreas Uhl. 2005. Image confidentiality using progressive JPEG. In *International Conference on Information Communications & Signal Processing*. IEEE, 1107–1111.
[36] Twitter. 2023. Twitter Image Pipeline (a.k.a. TIP). https://github.com/twitter/ios-twitter-image-pipeline, (accessed: Feb 13, 2023).
[37] W3Techs. 2022. Usage statistics of JPEG for websites. https://w3techs.com/technologies/details/im-jpeg (accessed: Jan 9, 2023).
[38] Gregory Wallace. 1992. The JPEG still picture compression standard. *Transactions on Consumer Electronics* 38, 1 (1992), xviii–xxxiv.
[39] Jiangtao Wen, M. Luttrell, and J. Villasenor. 2000. Trellis-based R-D optimal quantization in H.263+. *Transactions on Image Processing* 9, 8 (2000), 1431–1434.
[40] Andreas Westfeld. 2001. F5—a steganographic algorithm: High capacity despite better steganalysis. In *Information Hiding: 4th International Workshop, IH 2001 Pittsburgh, PA, USA, April 25–27, 2001 Proceedings*. Springer, 289–302.