ST. PÖLTEN UNIVERSITY
OF APPLIED SCIENCES

Informatik & Security

fh st.pölten

How synthetic images effects modern steganalysis

An experimental research of Cover Source Mismatch in colour JPEG domain

Master thesis

for attainment of the academic degree of

Diplom-Ingenieur/in

submitted by

Michael Anton Pristauz-Telsnigg

is231503

in the

University Course Information Security at St. Pölten University of Applied Sciences

Supervision

Advisor: Dipl.-Ing. Patrick Kochberger, BSc

Assistance: -

Declaration of Honour

First name, Surname: Michael Anton Pristauz-Telsnigg

Matriculation number: is231503

Title of the thesis: How synthetic images effects modern steganalysis

I hereby declare that

- I have written the work at hand on my own without help from others and I have used no other resources and tools than the ones acknowledged
- I have complied with the Standards of good scientific practice in accordance with the St. Pölten UAS' Guidelines for Scientific Work when writing this work.
- I have neither published nor submitted the work at hand to another higher education institution for assessment or in any other form as examination work.

Regarding the use of generative artificial intelligence tools such as chatbots, image generators, programming applications, paraphrasing and translation tools, I declare that

☐ no generative artificial intelligence tools were used in the course of this work.

☒ I have used generative artificial intelligence tools to proof-read this work.

☒ I have used generative artificial intelligence tools to create parts of the content of this work. I certify that I have cited the original source of any generated content. The generative artificial intelligence tools that I used are acknowledged at the respective positions in the text.

Having read and understood the St. Pölten UAS' Guidelines for Scientific Work, I am aware of the consequences of a dishonest declaration.

Abstract

The increasing prevalence of synthetically generated content, particularly in the form of deepfakes, poses new challenges for existing steganography analysis methods. This thesis deals with the analysis of deepfakes using steganalysis models in the colour JPEG domain, addressing in particular the problem of so-called cover source mismatch (CSM) - an effect that occurs when the training and test material comes from different sources. The open-source framework Aletheia is used for the investigation, which provides pre-trained models based on real image data. Two common steganography algorithms, J-UNIWARD and J-MiPOD, serve as the basis for embedding. The deepfakes are generated using modern diffusion models and compressed in three quality levels. The work takes an atomistic approach, applying existing Aletheia models to synthetic data and training and comparing fine-tuned models based on the generated deepfakes. The results clearly show that the pre-trained models on real data have significant shortcomings in the detection of deepfakes - this is particularly evident in low detection rates and high total error rates, often close to 50%, which suggests random classification. The primary cause lies in cover source mismatch. In contrast, the specially fine-tuned models on synthetic training data show significantly better performance, both in terms of detection rate and generalisation ability on foreign synthetic sources. In addition, it is observed that detection accuracy decreases with increasing JPEG quality of the images, while it increases with lower quality.

Contents

1. Introduction

In recent years, the rapid spread of artificial intelligence (AI) in a wide variety of areas of life has attracted considerable attention. AI systems provide support in areas such as image and speech processing, the healthcare sector and autonomous transport systems, only to name a few. These developments stand in stark contrast to traditional topics in computer science, such as steganography and steganananalysis - areas that have been studied for decades. Steganography deals with hiding information in a carrier medium, while steganography aims to detect such hidden information. Both disciplines exist in various domains and use different cover media - such as images, audio or video sequences.

In steganography analysis, the use of AI-based methods has increasingly replaced traditional algorithms. In recent years in particular, powerful models have been trained that surpass classical methods in terms of accuracy and automation. Traditional methods, such as classical feature extraction and ensemble classifiers, are increasingly falling behind.

The topic becomes interesting when not only the analysis models are AI-supported, but also the cover objects themselves are synthetically generated. While the focus has long been on heterogeneous real data sets, synthetic images are increasingly taking centre stage.

The increase in synthetic images on the internet, especially on social media or websites, and the growing availability of advanced generative models has led to such images also being used as cover media in steganography. Such synthetic images differ significantly from real images in terms of their statistical properties - both in terms of texture, compression and noise behaviour. It is therefore essential to investigate how the use of synthetic cover sources affects the effectiveness and accuracy of steganography analysis.

This work focuses on steganography analysis in the colour JPEG domain, considering four different synthetic image sources. These images are covertly modified using two state-of-the-art steganography algorithms. The focus on colour JPEG data is justified by the strong presence of this format in real-world application scenarios and already existing research and experience in the

spatial domain. In addition, JPEG is particularly relevant for resource-constrained platforms where storage optimisation is crucial due to its compression.

The aim of this work is to investigate the influence of the source (synthetic vs. real) on steganography analysis results. The following will be analysed:

- **How robust is a model trained on real JPEG images against cover-source mismatch when used for steganography analysis of artificially generated coloured JPEG images in different quality levels?**
- **How does fine-tuning new models in an atomistic approach using artificial data sets affect the prediction rate?**

Specific models trained on the heterogeneous real image datasets of the ALASKA-2 Challenge serve as a starting point. The extent to which this pre-trained models can be transferred to synthetic sources and the performance losses (cover source mismatch) that occur are investigated.

Previous research with synthetic sources in the spatial domain has already shown that AI-generated images (deepfakes) in the spatial domain are particularly difficult to examine for steganography analysis and cause significantly lower detection accuracy due to the pronounced cover source mismatch effect. Both holistic and atomistic analysis approaches reach their limits here, with detection performance depending heavily on the image generator used. [1]

1.1. Contribution

The following section lists the most important findings from this work:

- The models, trained on a real data sets, generalise very poorly when analysing synthetic images and suffers greatly from cover source mismatch.
- Different quality levels have an impact on detectability in steganography analysis. Low-quality stego images are easier to detect correctly than higher-quality images.
- Optimised models based on artificial data sets generalise significantly better than the models trained on the real data set, despite the atomistic approach.

With these findings, further research can be conducted in this area, algorithms can be recreated or optimised, and the effect of cover source mismatch can be further analysed.

1.2. Thesis Outline

The overall structure of this document is as follows:

- **Introduction**: The introduction (referenced as chapter 1) provides an overview of the topic, challenges, motivation, and the scope of the research.
- **Background**: This section (referenced as chapter 2) covers necessary prerequisites and fundamental knowledge relevant to the subject. Information about the JPEG standard, Cover Source Mismatch, steganography and steganalysis in general and other necessary topics are stated here.
- **Related Work**: The related work section (referenced as chapter 3) presents existing research and literature that pertains to our study.
- **Methodology**: In this part (referenced as chapter 4), we delve into the methodology employed in our research. The overall high level approach is stated here.
- **Implementation**: The implementation section (referenced as chapter 5) discusses the practical implementation of our approach. All steps made in the research are documented here. Also code snippets and dicisions are explained.
- **Results**: In this section, we report the results of our experiments in the section labeled chapter 6.
- **Discussion**: Than we discuss and interpret the results of our experiments in context of our reserach questions in the section labeled chapter 7.
- **Conclusion**: Finally, we draw conclusions and summarize the key findings and contributions of this thesis in chapter 8.

2. Prerequisites

The chapter outlines the basic requirements necessary for understanding and implementing this work. These include, in particular, the theoretical foundations, technologies used, and relevant framework conditions that serve as the basis for further explanations. The aim is to create a common understanding and to make the technical and methodological requirements transparent.

2.1. Steganography

The idea of concealed writing is an old one: Herodotus (5th century BC) reports of warning messages hidden under a layer of wax on wooden tablets; Aeneas Taktikos used similar methods. In the 20th century, spies perfected the microdot technique, in which miniaturised microfilm images of entire documents appeared as apparent dots in letters and were used in both world wars. [2] [3]
A theoretical framework was first provided by G. J. Simmons' Prisoners' Problem in 1983, which demanded that messages be encapsulated in such a way that even an omniscient guard could not prove their existence. [2]
Image steganography refers to the insertion of a secret payload into digital images, whereby the modifications are so minor that neither the presence of the message nor its content is noticeable; in contrast to cryptography, the communication itself is concealed and is part of steganography. [2]
Image steganography is part of a larger family: related techniques hide data in audio signals, videos, texts or even network protocol fields; what they all have in common is that they exploit transmission media with sufficient redundancy to lower the message below the noise threshold of the respective perception system. [4]
The spatial domain is particularly popular for digital images. The least significant bit (LSB) method replaces the least significant bits of the RGB channels with the message; typically, three bits can be accommodated per pixel without any perceptible colour change. A simple example shows that changing from 10110100 to 10110101 only raises the red value from 180 to 181 and remains

visually unnoticeable. [5]

Research shows practical payloads of around 3 bits per pixel (bpp) before statistical artefacts reveal the image, with modern LSB variants even exceeding this thanks to hash dispersion or error correction. [6] [7]

In order to survive lossy compression - especially JPEG - transformation methods shift the embedding into the frequency domain. After the discrete cosine transform (DCT) of an 8 x 8 block, the middle frequency coefficients are preferably changed minimally: changes in low frequencies would be immediately visible, while high frequencies are usually lost during quantisation. The middle ground chosen in this way therefore offers a good balance between inconspicuousness and consistency; even after multiple JPEG compressions, the hidden data can be reliably reconstructed. [8] [9]

2.1.1. Spatial versus JPEG domain

In digital image steganography, there are two main areas of work: the spatial domain (also known as the pixel or space domain) and the JPEG or transform domain. In the spatial domain, the grey or colour values of the image pixels are changed directly, while transform methods first perform a frequency-based decomposition - typically the discrete cosine transform (DCT) in the JPEG codec - and then modify selected coefficients. Even basic overviews emphasise that this allows three core parameters to be determined: payload (capacity), robustness against image processing and detectability by steganography. [10]

The classic spatial method is least significant bit (LSB) substitution. By overwriting only the least significant bit of a pixel, the visual impression remains virtually unchanged. Current experiments with 165 RGB test images achieve peak values of over 50 dB PSNR, which corresponds to virtually lossless image quality. [11]

However, this high payload comes at the price of low resilience: even simple JPEG compression, noise injection or the swapping of individual bits can destroy the hidden message or reveal it statistically. Fridrich, Goljan & Hogea show that flipping just 10% of the usable DCT coefficients in the F5 context is reliably detectable and that even individual LSB modifications can be detected. [12]

Transform-based approaches shift the embedding into the frequency domain. After performing the 8 x 8 block DCT, mainly mid-range frequencies are manipulated because changes there are inconspicuous to the human eye but comparatively stable for lossy quantisation. Algorithms such as JSteg or the more advanced F5 use matrix coding to reduce the embedding space on the one

hand and to minimise the number of coefficients that are actually changed on the other, thereby reducing statistical anomalies - a strategy that has become vulnerable again with better histogram estimators, however. [12]
Recent research combines the JPEG domain with learning-based robustness classifiers. Zhang, Zhao & He train a neural model on typical social media recompression channels and select only those DCT blocks for embedding that have a high probability of survival; this significantly increases the extraction rate without drastically reducing the payload. [13]
Nag et al. pursue a different optimisation: By Huffman encoding the ciphertext before embedding, the data can be packed more densely, allowing more information to be transmitted at the same level of visibility. At the same time, the structure of the DCT coefficients remains largely untouched, which increases visual inconspicuousness. [14]
A direct comparison reveals a clear conflict of objectives. Spatial methods offer very high capacity and are algorithmically simple, but are extremely susceptible to lossy processing and are comparatively easy to detect. Transform methods sacrifice capacity in favour of greater robustness and fewer visual artefacts; their implementation is more complex and computationally intensive, but they withstand common JPEG quality factors or platform recompressions much better. Practice-oriented guidelines therefore recommend LSB methods only in controlled, lossless channels, while public or highly compressive environments clearly favour the JPEG domain.

2.2. JPEG standard

This chapter and its subchapters explain JPEG compression in sufficient detail to provide a basic understanding of the standard and its individual steps. This knowledge is essential for further research in order to comprehend the processes, decisions and results and to understand the actual focus of this work. First, a summary of Wallace's paper [15] on the standard, which was published at the time, is provided. This contains information on the history and considerations behind the standard, as well as technical procedures. After this introduction, the individual technical process steps are broken down as simply as possible.

2.2.1. History and summary of the standard

JPEG defines the first international standard for compressing continuous-tone still images, supporting both grayscale and color formats. It introduces a flexible and generic framework intended

to serve a wide range of applications, including digital photography, medical imaging, desktop publishing, and facsimile transmission. The standard supports both lossy and lossless compression modes to balance image quality, compression efficiency, and computational complexity. At the core of JPEG's lossy compression is the Discrete Cosine Transform (DCT). An image is divided into 8x8 blocks of pixel values, each block undergoing a DCT to convert spatial domain data into frequency domain coefficients. This transformation concentrates energy in the lower spatial frequencies, allowing high-frequency components - often visually negligible - to be discarded or quantized coarsely. The DCT coefficients are quantized using a customizable quantization table, which introduces most of the loss in compression by reducing coefficient precision. The quantized values are then entropy coded, most commonly using Huffman coding in the Baseline mode. The result is a significant reduction in data volume, often achieving compression ratios of 10:1 to 50:1 without perceptible quality degradation. JPEG defines four modes of operation: sequential, progressive, lossless, and hierarchical. The sequential mode encodes each component in a single scan from top-left to bottom-right. Progressive mode allows for multiple scans, starting with a coarse version of the image and gradually refining it through spectral selection and successive approximation. This is particularly useful for slow transmission channels where early previews are valuable. The hierarchical mode encodes the image at multiple resolutions, enabling quick access to lower-resolution versions without full decompression. Lossless compression is achieved through predictive coding, where each pixel is predicted from its neighbors, and the residual is entropy coded. Although compression ratios are lower, the method ensures exact reconstruction. Color images are treated as multi-component data, typically with chrominance subsampling. JPEG supports component interleaving, allowing data from multiple channels (e.g., YCbCr) to be encoded together in units called Minimum Coded Units (MCUs). Sampling ratios can differ between components, and the format accommodates up to four interleaved components per MCU. The structure of MCUs and interleaving strategies enables efficient decoding and pipelining in practical systems. Quantization and entropy coding tables must be specified by the application or embedded in the compressed image file. JPEG does not mandate specific tables, allowing implementations to optimize based on image content and desired quality. However, compliance requires that decoders handle externally defined tables and achieve a minimum accuracy in DCT calculations. Typical compressed color images achieve visually good quality at bitrates of 0.5-0.75 bits per pixel and near-lossless quality above 1.5 bits per pixel. Applications requiring high fidelity, such as medical imaging, benefit from the 12-bit precision support provided in JPEG's extended

modes. By standardizing both the encoding syntax and interchange format, JPEG enables reliable transmission and storage of images across diverse hardware and software platforms. It ensures interoperability, encourages hardware acceleration through VLSI integration, and has become foundational to digital imaging workflows. [15]

2.2.2. Technical steps of JPEG compression

JPEG compression consists of a chain of sequential steps, each of which specifically removes redundancy or information that is less important to the eye. In the following, the reader will be guided step by step through the classic baseline process.

Colour space transformation First, the image is converted from the RGB colour space to the YCbCr colour space. In this new colour space, the brightness information (luminance, Y) is separated from the colour information (chrominance, Cb and Cr). This step is based on the fact that the human eye is much more sensitive to differences in brightness than to changes in colour. The transformation is performed by linearly weighting the RGB components, resulting in a more efficient separation of relevant image content for later compression.

Chroma subsampling After the colour space transformation, the chrominance channels Cb and Cr are subsampled. This reduces the resolution of these colour channels, usually in a ratio of 4:2:0 or 4:2:2. This means that only one or two chrominance values are stored for every four luminance values. This step takes advantage of the fact that fine colour details are less relevant to the human eye, which allows for a reduction in the amount of data with minimal perceptible loss of quality.

Division Into 8 x 8 Pixel Blocks The image is then divided into blocks of 8 x 8 pixels. Each of these blocks is processed independently, which simplifies the application of the following mathematical transformations and makes them more efficient. This block size is a compromise between compression efficiency and computing power and is a central element of the JPEG standard.

Forward DCT (Discrete Cosine Transform) A two-dimensional discrete cosine transform is applied to each 8 x 8 block. This transform converts the pixel values in the spatial domain into frequency components in the frequency domain. The majority of the image information is concentrated in the low-frequency coefficients, while high-frequency components usually represent

fine image details or noise. The DCT thus serves to compress energy by bringing the essential information to the beginning of the coefficient matrix.

Quantization The resulting DCT coefficients are quantised in the next step. Each coefficient is divided by an entry in a standardised or user-defined quantisation table and then rounded. This reduction in numerical accuracy leads to the loss of irrelevant image information, especially in the high frequency range. Quantisation is the only lossy step in the JPEG compression process and is largely responsible for reducing the file size.

Zickzack scan After quantisation, the coefficients are read out in a zigzag order. This special order starts at the upper left corner of the matrix and follows a diagonal path through the values. The aim of this procedure is to group together the frequently occurring zero values, which are typically found at the end of the coefficient matrix. This makes it easier to apply the subsequent entropy coding.

Entropy encoding The final step in JPEG compression is the lossless encoding of the quantised and zigzag-sorted coefficients. First, run-length encoding (RLE) is used to efficiently represent longer sequences of zeros. This is followed by Huffman encoding, in which frequently occurring values are encoded with shorter bit sequences than rare ones. Alternatively, arithmetic encoding can also be used, although Huffman is more common in the JPEG standard. This combination results in a further significant reduction in the amount of data without any further loss of quality.

2.3. J-MiPOD explained

[1]

The following subchapter explains the J-MiPOD algorithm. It covers the background, motivation, and implementation.

2.3.1. Context and Motivation

Modern adaptive steganography assigns to every host element (pixel, DCT coefficient, ...) a *change-probability* β and tries to choose these probabilities so that a chosen detector cannot

[1] 2.3 created with help of ChatGPT o3 and ChatGPT 4o based on [16]

distinguish cover and stego objects. Most popular methods-HILL, S-UNIWARD, UERD, ...-build an *intuitive cost map* and then minimise the *average cost*. MiPOD and its JPEG counterpart J-MiPOD take a radically different, *model-driven* route:

- **Assume** a realistic statistical model of covers.
- **Write down** the *most powerful detector* (omniscient likelihood-ratio test).
- **Choose** the probabilities β that *minimise the detector's asymptotic power* subject to a payload constraint.

Thus the embedding problem becomes a well-posed optimisation problem whose objective is the detector's *deflection coefficient*. This is the core idea behind both MiPOD (spatial domain) and J-MiPOD (JPEG domain).

[16]

2.3.2. Spatial MiPOD

Cover model For every pixel[2]

$$x_{k,l} \sim \mathcal{N}\big(\theta_{k,l}, \sigma^2_{k,l}\big), \tag{2.1}$$

where the variances $\sigma^2_{k,l}$ vary across the image (heteroscedastic shot-noise model).

[16]

Most powerful test and deflection coefficient Under ternary ± 1 embedding the log-likelihood ratio of the omniscient detector simplifies (after a Taylor expansion) to an *additive* statistic whose mean shift is

$$\Delta\mu \;=\; 2\sum_{k,l} \beta^2_{k,l}\, \sigma^{-4}_{k,l}. \tag{2.2}$$

Its (unit) variance gives an asymptotic $\mathcal{N}\big(\sqrt{2\mathcal{D}}, 1\big)$ distribution under H_1, where

$$\boxed{\mathcal{D} \;=\; \sum_{k,l} \beta^2_{k,l}\, \sigma^{-4}_{k,l}} \tag{2.3}$$

is the *deflection coefficient*. The smaller $\mathcal{D}$, the harder the optimal test.

[16]

[2] Indices k, l run over image coordinates and will be suppressed when unambiguous.

Optimisation problem Let the (relative) payload be the *ternary entropy*

$$R \;=\; \sum_{k,l} H_3\big(\beta_{k,l}\big), \qquad H_3(x) = -2x\log_2 x - (1-2x)\log_2(1-2x). \tag{2.4}$$

MiPOD solves[3]

$$\min_{\{\beta\}} \sum \beta^2 \sigma^{-4} \quad \text{s.t.} \quad \sum H_3(\beta) = R. \tag{2.5}$$

With a Lagrange multiplier λ,

$$f(\beta) \;=\; \beta \log \frac{1-2\beta}{\beta} \quad \Longrightarrow \quad \beta = f^{-1}\big(\lambda\,\sigma^4\big), \tag{2.6}$$

where f^{-1} is found once and tabulated. λ is obtained by a scalar bisection so that the payload constraint is met.

[16]

Turning probabilities into costs Actual embedding uses syndrome-trellis codes (STCs). The cost that reproduces 2.6 is

$$\rho \;=\; \frac{1}{\lambda}\,\ln\big(\tfrac{1}{\beta} - 2\big). \tag{2.7}$$

[16]

Quick numerical toy example Two pixels, variances $\sigma_1^2 = 1$, $\sigma_2^2 = 4$; payload $R = H_3(\beta_1) + H_3(\beta_2) = 0.1$ bpp. Solving (2.5) yields $\beta_1 \approx 0.006$, $\beta_2 \approx 0.017$: *the "noisier" pixel carries ∼3× more hidden data*, exactly what intuition suggests.

[16]

2.3.3. J-MiPOD (JPEG domain)

Additional ingredients After 8×8 DCT each coefficient $c_{m,n}$ is quantised with step $\Delta_{m,n}$:

$$\bar{c}_{m,n} = \mathrm{round}\big(c_{m,n}/\Delta_{m,n}\big).$$

Assuming the quantiser is fine relative to the noise ($\sigma_{m,n} \gg \Delta_{m,n}$), the pmf of $\bar{c}_{m,n}$ is well approximated by

$$P\big[\bar{c} = k\big] \;\approx\; \frac{\Delta_{m,n}}{\sigma_{m,n}}\,\varphi\Big(\frac{k\Delta_{m,n} - \theta_{m,n}}{\sigma_{m,n}}\Big),$$

with φ the standard normal pdf.

[16]

[3] Indices are suppressed for clarity.

Deflection & optimisation For JPEG the deflection becomes

$$\boxed{\mathcal{D} \;=\; \sum_{m,n} \beta_{m,n}^2 \,\big(\tfrac{\Delta_{m,n}}{\sigma_{m,n}}\big)^4} \tag{2.8}$$

and the optimisation is

$$\min_{\{\beta\}} \sum \beta^2 \big(\tfrac{\Delta}{\sigma}\big)^4 \quad \text{s.t.} \quad \sum H_3(\beta) = R. \tag{2.9}$$

Solution:

$$\beta_{m,n} = f^{-1}\Big(\lambda\, \sigma_{m,n}^{-4}\, \Delta_{m,n}^4\Big), \tag{2.10}$$

with the *same* f as in MiPOD. Hence J-MiPOD differs from MiPOD only by the scale factor $(\Delta/\sigma)^4$- large quantisation steps receive smaller probabilities even when their variance is high.
[16]

2.3.4. What makes (J)MiPOD different?

- **Detector-centric design.** Most methods guess a cost, then test against many steganalysers. MiPOD starts with the *optimal* one and guarantees that *no* other detector can do better under the adopted model.
- **Closed-form probabilities.** Once σ (and Δ for JPEG) are estimated, (2.6) gives β directly, no heuristic tweaking.
- **Smooth adaptation.** Because $\beta \propto \sigma^4$, probabilities vary smoothly with local noise level; this avoids the "over-adaptive" behaviour of some cost-based schemes and greatly improves robustness against deep-learning steganalysis.
- **Side-informed extension.** When the uncompressed image ("pre-cover") is available, J-MiPOD can constrain modifications to the direction that *minimises* the physical change ($\pm\Delta(1/2 - e)$ where e is the quantisation error), simply by replacing Δ in (2.8) with the actual magnitude of that change.

[16]

2.3.5. Practical hints

- **Variance estimation.** A fast 2×2 Wiener filter followed by a local polynomial fit ($q = 3$, degree $d = 3$) gives reliable σ^2 while keeping textured content in the residual (crucial for security).

- **Implementation speed.** For a 512×512 JPEG, reference MATLAB code needs ≈ 0.4 s (no parallelism)-4-8$\times$ faster than UNIWARD and only slightly slower than UERD.
- **Coding.** Feed the costs (2.7) to an $n = 7$ syndrome-trellis code to achieve $\gtrsim 98\%$ of the theoretical payload.

[16]

2.3.6. Take-away example (JPEG)

Assume one 8×8 block with two AC coefficients

	coeff. 1	coeff. 2
σ	3	1
Δ	2	1

Compute the *embedding weights* $\left(\frac{\Delta}{\sigma}\right)^4$: $w_1 = (\frac{2}{3})^4 = 0.20,\ w_2 = 1^4 = 1$. Even though coefficient 1 is "noisy", its large quantiser makes it five times *less* attractive than coefficient 2-a choice traditional cost maps rarely make but which follows directly from (2.8).

[16]

2.3.7. Conclusion

MiPOD and J-MiPOD replace hand-crafted heuristics with a clean, statistically optimal objective: *minimise the power of the best possible detector*. The resulting probabilities (2.6)-(2.9) adapt naturally to local noise and quantisation, leading to state-of-the-art security *especially against deep neural steganalysers*, while staying computationally lightweight.

[16]

2.4. UNIWARD and J-UNIWARD explained

[4]

The following subchapter explains the J-UNIWARD algorithm. It covers the background, motivation, and implementation.

[4] 2.4created with help of ChatGPT o3 and ChatGPT 4o based on [17]

2.4.1. Overall Focus

UNIWARD (*UNiversal WAvelet Relative Distortion*) is a family of steganographic distortion functions whose *only* goal is to assign a cost to every potential modification of a digital image so that the message can be embedded where changes are statistically hardest to detect. Its key principles are:

- **Universality**: the *same* mathematical expression drives embedding in raw pixels (spatial images), in JPEG images, and even in side-informed JPEG.[5]
- **Directionality**: changes are evaluated in three undecimated first-level wavelet sub-bands (horizontal, vertical, diagonal), so edges that can be modelled in *any* direction receive a high penalty, while unpredictable texture receives a low penalty.
- **Relativity**: the penalty is *relative* to the magnitude of the local wavelet coefficient, so disturbing strong texture costs little, disturbing a flat region costs a lot.

J-UNIWARD (often written *JUNIWARD*) is simply *UNIWARD applied to JPEG images*. Because a JPEG file stores quantised DCT coefficients, the algorithm first *decompresses* the image to the spatial domain and then computes exactly the same wavelet-relative distortion. Hence:

$$D_{\text{JUNIWARD}}(X, Y) \;=\; D\big(J^{-1}(X),\, J^{-1}(Y)\big) \qquad \text{(see Eq. (4))}$$

This trick avoids designing a brand-new JPEG-specific model, yet automatically discourages changes in high-frequency AC coefficients (they appear almost always quantised to zero after decompression).

[17]

2.4.2. Mathematical Core

Directional filter bank Choose 1-D wavelet analysis filters h (low-pass) and g (high-pass). Build three 2-D kernels (Eq. (2))

$$K^{(1)} = h\, g^{\top}, \quad K^{(2)} = g\, h^{\top}, \quad K^{(3)} = g\, g^{\top}.$$

For a cover image X these kernels give directional residuals $W^{(k)} = K^{(k)} \star X$ (mirror-padded convolution).

[17]

[5] The universal idea is introduced in Section 3 of the original paper.

Universal distortion (spatial images) For a candidate stego image Y the *non-additive* distortion is (Eq. (3))

$$D(X,Y) \;=\; \sum_{k=1}^{3}\sum_{u=1}^{n_1}\sum_{v=1}^{n_2} \frac{\left|W_{uv}^{(k)}(X) - W_{uv}^{(k)}(Y)\right|}{\sigma + |W_{uv}^{(k)}(X)|},$$

where $\sigma > 0$ only stabilises division. A large wavelet coefficient in *any* direction lowers the denominator, so noisy texture is cheap, flat regions are expensive.

[17]

JPEG version Let X and Y be the *arrays of quantised DCT coefficients*. Decompress, then reuse the same formula:

$$D_{\text{JUNIWARD}}(X,Y) = D\big(J^{-1}(X), J^{-1}(Y)\big). \tag{4}$$

[17]

Additive approximation for practical coding Instead of optimising Eq. (3) directly, UNIWARD uses a pixel-wise cost

$$\rho_{ij}(X,y) = D\big(X,\, X_{\sim ijy}\big), \qquad X_{\sim ijy}:\; X_{ij} \leftarrow y$$

and embeds close to the *payload-distortion bound* with syndrome-trellis codes. The final additive distortion is

$$D_A(X,Y) = \sum_{i,j} \rho_{ij}(X, Y_{ij})\,[X_{ij} \neq Y_{ij}], \tag{7}$$

where the Iverson bracket equals 1 when the pixel was altered, 0 otherwise.

[17]

2.4.3. How UNIWARD Differs from Earlier Schemes

HUGO: penalises changes that alter a rich model computed on *pixel differences*. UNIWARD skips explicit modelling, using the simpler relative-wavelet idea and thus generalises to JPEG.

WOW: also uses three directional sub-bands but aggregates them with a Hölder norm. Extending WOW to JPEG is tricky; UNIWARD is plug-and-play across domains.

UED / nsF5 (JPEG): work *inside* the DCT domain and require hand-crafted heuristics (e.g. ignore zeros). UNIWARD simply views the JPEG through spatial wavelets; zeros in high-frequency bands are naturally expensive, giving better empirical security.

[17]

2.4.4. Worked Examples

Spatial 4 x 4 toy patch Assume a 4×4 cover block X whose horizontal high-pass residual at position $(2, 3)$ equals $W^{(1)}_{2,3}(X) = 24$. We consider changing pixel $X_{2,3}$ by $+1$, producing Y with $W^{(1)}_{2,3}(Y) = 22$ (all other residuals unchanged) and $\sigma = 1$.

$$\rho_{2,3}(X, X_{2,3} + 1) = \frac{|24 - 22|}{1 + |24|} = \frac{2}{25} = 0.08.$$

If instead we tried the same change in a flat area where $W^{(1)}_{uv}(X) = 2$, the cost would be $2/(1+2) = 0.67$ - more than eight times higher. So the encoder will prefer the textured site.

[17]

JPEG 8 x 8 block Let $X_{pq} = 0$ be a high-frequency AC DCT coefficient (quantisation step $q_{pq} = 9$). After decompression $J^{-1}(X)$ contributes essentially nothing to any wavelet band, so $|W^{(k)}_{uv}(X)| \approx 0$. Flipping its LSB would decompress to ± 9, giving a large numerator but a tiny denominator, hence a very *high* cost. Conversely, altering a mid-frequency coefficient already equal to $7q_{pq}$ leaves the denominator large and the relative change small, so the cost drops. Thus J-UNIWARD automatically *avoids* empty high-frequency bins without extra rules.

[17]

2.4.5. Practical Take-aways

- Use $\sigma = 1$ in the spatial domain and $\sigma = 2^{-6}$ in JPEG for best security.
- A Daubechies-8 filter bank offers the best resistance to rich-model steganalysis.
- Embedding is implemented with ternary (spatial/JPEG) or binary (side-informed JPEG) syndrome-trellis codes; coding loss is negligible.

Summary: UNIWARD minimises *relative* wavelet distortion, adapting to unpredictable texture in *any* direction and carrying over unchanged to JPEG images (J-UNIWARD) by a simple spatial

back-projection. This universality, combined with efficient coding, explains its consistently superior empirical security compared with earlier content-adaptive schemes.
[17]

2.5. Cover Source Mismatch

In steganalysis, cover source mismatch (CSM) refers to the scenario in which a detector is trained on cover images (or audio, etc.) from a specific source (cover source) but must later be applied to objects from a different source. Because even small differences in the camera sensor, ISO value, white balance, scaling algorithms or even a second JPEG compression are reflected in the statistics of the image features, the underlying probability distribution P(x) changes. This leads to a distribution shift, which often causes the error rate to rise from single-digit percentages to random hits (approximately 50%). [18] [19]
Historically, CSM was first described around 2010, when it was discovered that a classifier trained on images from camera CAM1 performed significantly worse on images from camera CAM2; in the 2010 BOSS competition, additional test images with a different recording chain completely knocked out the recognition rate. Later work showed that not only camera models, but also different image processing pipelines, double compression or synthetically generated content increase the mismatch gap. [19]
CSM can be formalised by considering the training $P_{\text{trn}}(x)$ and testing $P_{\text{tst}}(x)$ as two source distributions and measuring their degree of divergence using the Kullback-Leibler divergence

$$D_{\mathrm{KL}}(P_{\text{tst}} \parallel P_{\text{trn}}) = \sum_i P_{\text{tst}}(i)\, \log \frac{P_{\text{tst}}(i)}{P_{\text{trn}}(i)}$$

or about changing the classification error

$$\Delta P_E \;=\; P_E^{\text{CSM}} - P_E^{\text{in-source}}$$

. The larger D_{KL} or ΔP_E, the stronger the CSM effect. Current analyses rate CSM as one of the biggest obstacles in the transition from laboratory steganography to operational systems. [20]
Countermeasures can be roughly divided into two schools of thought:

1. Atomistic means clustering the entire data set into sub-sources that are as homogeneous as possible and training a special detector for each source. This reduces the divergence term, but requires source recognition in the field.

2. Holistic, on the other hand, collects an extremely diverse training set in order to learn a robust model; deep learning architectures show a certain, but by no means complete, resilience here. Newer approaches combine both, modelling CSM as a robust optimisation problem or using domain-specific fine-tuning to bridge the distribution gap with few additional images. [19] [20]

3. Related Work

In recent years, research in the field of digital image steganography has evolved from specific heuristic methods to data-driven, statistically sound and machine learning-based methods. A key problem that has been exacerbated by this development is that of so-called cover source mismatch (CSM). This refers to the significant reduction in the detection performance of steganography analysis methods when the statistical properties of the training and test images do not match - for example, due to different camera sensors, post-processing, compression algorithms or colour spaces. In the following sections, research already done in the fields of CSM, usage of AI driven analysis methods and the usage and analysis of synthetic sources, will be stated.

3.1. Research on Cover Source Mismatch

A central problem in modern image steganography analysis is the so-called cover-source mismatch (CSM), which occurs when training and test images come from different sources, i.e., are subject to different statistical distributions. This phenomenon was first documented in the work of Cancelli et al. and in the context of the BOSS competition, and has since emerged as a key obstacle to the practical application of steganography techniques. [21]

Giboulot et al. (2020) systematically analysed various factors that can cause CSM, including camera models, image processing procedures and JPEG compression. Their studies showed that image development parameters in particular, such as sharpening, noise reduction and the choice of quantisation table, have a significant influence on the effectiveness of steganography detection methods. In addition, two metrics were introduced: source difficulty and source inconsistency, to quantify the effects of CSM. As solution strategies, the authors discussed the targeted selection of training data (specific vs. diversified) and the use of a pre-classifier for source identification. [21] [22]

Sepák et al. (2022) proposed a formal definition of CSM based on a robust optimisation approach.

Their work highlights that previous approaches to mitigating CSM are based either on the atomistic strategy - i.e., training multiple specific detectors for different sources - or the holistic strategy, in which a detector is trained on a broad mix of sources. They show that neither of these strategies eliminates all disadvantages and propose alternative metrics for error evaluation. [23]

In an in-depth analysis of the influence of CSM on deep learning-based steganography analysis, Giboulot et al. (2022) show that even modern neural networks such as SRNet and EfficientNet suffer greatly from CSM, even though they theoretically have better generalisation capabilities. Nevertheless, the holistic training strategy shows certain advantages in combination with deep learning, as it can better exploit the generalisation potential, even with smaller training sets. [24]

A novel approach was proposed by Méreur et al. (2024), which investigates the use of deepfakes as cover images. It shows that synthetically generated images are particularly suitable for exploiting the CSM problem in the interests of steganographers due to their high variability. Deepfakes can be specifically designed to undermine the training assumptions of detectors. These results illustrate that the CSM problem will continue to gain relevance in future applications, especially in the context of generative AI. [1]

In parallel, a recent paper [25] by Mallet et al. (2024) addresses the influence of source mismatch on deepfake detection systems. Here, too, it is shown that models trained on a specific deepfake source (e.g., a special text-to-image model with fixed hyperparameters) lose their performance dramatically when confronted with images from other sources or slightly different generation parameters. This analogy to CSM in steganography analysis illustrates that the underlying problem of distribution shift is cross-domain and represents one of the biggest challenges for the transfer of detection methods to real-world applications. [25]

This summary of the literature illustrates that cover-source mismatch is a major obstacle to robust, generalising steganography analysis methods. Despite intensive research, there is still no universal approach that works reliably under real-world conditions. The identified solutions - atomistic and holistic strategies as well as deep learning methods with broad training - offer promising approaches, but need to be further refined and systematically evaluated in future research.

3.2. Research on cover source mismatch in the JPEG domain

Another important area of research in steganography deals with the causes, effects and mitigation possibilities of cover source mismatch (CSM) in JPEG-based image processing environments. In

particular, several studies have investigated the influence of specific image processing processes and the JPEG implementation used on detection accuracy.

Boroumand and Fridrich (2017) address the challenge of recovering the processing history of JPEG-compressed images. They developed a scalable method for classifying global image processing operations based on rich models. The methodology involves classification in the space of projected features, enabling multivariate Gaussian modelling of each processing class. This makes it possible to detect individual operations such as softening, denoising, sharpening and tone adjustment with high accuracy, even in JPEG-compressed images. It is interesting to note that JPEG compression levels out the differences between sources, which slightly reduces the influence of CSM compared to uncompressed images. [26]

Benes et al. (2022) made an experimental contribution to the CSM problem by quantifying the influence of the JPEG implementation (libjpeg version 6b vs. 7) on the CSM error rate. The work shows that the JPEG library used alone causes measurable differences in the feature space during compression and decompression. These differences are sufficient to significantly impair detection performance in a J-UNIWARD/JSRM+EFLD scenario. The use of different libjpeg versions between training and test data is particularly critical - a previously underestimated but reproducible influencing factor on the CSM problem. [27]

In summary, recent literature consistently shows that cover-source mismatch can be triggered by classic factors such as camera model and image processing, as well as by technical details such as JPEG implementation and library version. Robust and generalisable steganography analysis therefore requires not only powerful classifiers, but also careful consideration of source diversity and adaptive training strategies. The trend is increasingly towards hybridised methods that combine rich features, deep learning and probabilistic modelling to effectively counteract the effects of CSM in real-world applications.

3.3. Delimitation and contribution of this work

This work builds on the research [1] of Méreur et al., but expands the existing research framework in several key aspects. While the aforementioned work focuses exclusively on synthetic, uncompressed grayscale images, this paper conducts an investigation in the *JPEG colour image domain*, i.e. in a more realistic and technically demanding context. In particular, the effects of JPEG compression on CSM and steganalysis performance are systematically evaluated - an aspect that has

hardly been addressed in previous work.

In addition, this work uses the open-source tool *Aletheia*, which allows pre-trained models to be accessed and their DCI values and prediction scores to be analysed. This integration not only allows for transparent evaluation, but also bridges the gap between academic theory and practical application. [28]

Instead of relying on a large number of generators, a controlled focus is placed on four common diffusion models, from each of which 10,000 images were generated. This deliberate reduction enables a targeted analysis of the internal image structure and allows changes in detection performance to be precisely recorded when switching from training to test sources.

Another focus is on comparing recognition rates before and after targeted re-training on proprietary deepfake datasets. This not only tests the generalisability of existing models, but also examines their ability to learn from new sources.

Last but not least, the current state of hardware - especially affordable GPUs - enables the training of complex models such as EfficientNet even in resource-limited research scenarios. This opens up new empirical possibilities that were not available in earlier work.

This work makes an independent contribution to steganography research by transferring insights from the deepfake domain to a compressed, colourful and operationally relevant application context, using modern tools and evaluation metrics that ensure the transferability and reproducibility of the results.

4. Methodology

4.1. Introduction and overall methodological strategy

The following subchapter describes the motivation for the method used and provides a brief summary of what is done and how.

4.1.1. Motivation for using this method

A combination of literature research and experimental research is used. The literature research provides a good overview of previous research, its findings, limitations and open questions. This made it possible to derive a specific research topic. Experiments can then be used to test the open questions or the derived theses/research questions.

4.1.2. What and How?

This thesis investigates the influence of cover source mismatch (CSM) on the performance of a widely used, publicly available steganography analysis tool that enables steganography analysis using pre-trained models. Artificially generated image datasets are used as input. Among other things, the tool provides ready-trained models based on known real image datasets and also allows users to train their own models based on EfficientNetB0.

The results obtained with the tool provide insights into how strongly CSM affects detection results, whether both pre-trained and self-trained models are suitable for analysing artificial datasets, and to what extent the results differ when models are trained specifically on artificially generated data.

Various experiments are conducted to answer these questions.

4.2. Research design

The following subchapter lists the research questions and hypotheses, describes the choice of research approach, and defines the variables used.

4.2.1. Research questions and hypotheses

1. **How robust is a model trained on real JPEG images against cover-source mismatch when used for steganography analysis of artificially generated coloured JPEG images in different quality levels?**
   To answer this question, pre-trained EfficientNetB0 models are used to analyse data sets from four diffusion models, each in three different quality levels after JPEG compression. For comparison, a real data set similar to the training data set is used. Several metrics are used to determine the effect of cover source mismatch. The exact implementation of the experiments and the evaluation can be found in the chapter 5.
2. **How does fine-tuning new models in an atomistic approach using artificial data sets affect the prediction rate?**
   To answer this question, EffnetB0 models in Aletheia [28] are fine-tuned with artificial data sets generated by diffusion models and then analysed. The difference in the results compared to the results from the first research question answers this question. The implementation of the corresponding experiment can be found here: 5.1.4

4.2.2. Choice of research approach

A combined research approach consisting of literature research and experimental investigation is chosen for this work.
The comprehensive literature research serves to systematically prepare the current state of research in the areas of cover source mismatch, AI-based steganography analysis, steganography analysis of deepfakes using AI, and steganography analysis in the JPEG domain. Work that deals with a combination of these topics is also taken into account. The research questions of the thesis are derived on the basis of this analysis.
This is followed by an experimental investigation. The aim is to collect quantitative results and metrics that can be compared with existing work. These metrics (see detailed description later in this chapter) cannot be meaningfully estimated, but must be calculated empirically by means of

experiments. A purely qualitative derivation would only allow for vague statements and would not be sufficiently precise from a scientific point of view.
The experiments are partly based on existing studies from the literature, which are summarised and referenced in Chapter 3.

4.3. Data sets

The following subchapter describes the data sets used and generated and explains their origin. It also describes the models used to generate the deepfakes.

4.3.1. Data types and sources

For this work, four datasets consisting of artificially generated images and one real dataset consisting of real images are processed.

Alaska2 real dataset

The ALASKA 2 dataset comprises a total of 80,000 raw images taken with more than forty different camera models - from smartphones and inexpensive compact cameras to professional full-frame DSLRs. All images were processed using deliberately diverse development workflows in order to replicate the heterogeneity of real Internet images and thus promote research approaches that generalise beyond strictly controlled laboratory conditions. A prepared subset is available for supervised steganography tasks: four training packages with a total of 75,000 carrier images and a separate evaluation set with 5,000 images. All files are available as coloured JPEGs with quality factors of 95, 90 and 75. For each carrier, there are three steganography versions generated using the J-UNIWARD, UERD and J-MiPOD methods; The average embedding load is 0.4 bits per non-zero DCT coefficient and adapts to the image complexity. In addition to these JPEG sets, the platform provides both the complete raw data archive and several pre-processed variants. Uncompressed colour and greyscale images are offered in resolutions of 512 x 512 px and 256 x 256 px; in addition, there are JPEG versions of the same dimensions with quality levels of 100, 95, 90, 85, 80 and 75. The conversion from the raw files to the distributed versions is documented by publicly available Python scripts based on NumPy, Pillow and RawTherapee, so that researchers can easily reproduce their own subsets or alternative export paths. All materials are subject to

the Creative Commons BY-NC-ND licence; they may therefore be used freely for non-commercial research and teaching, but require attribution and prohibit commercial use and the distribution of derivative works without the prior permission of the authors. [29]
This work uses the raw data set consisting of 80,000 colour images in 512x512 format. Except for the format cropping, the images are virtually raw and unchanged. The images are delivered in .tiff format.

Artificially generated datasets

The artificial image datasets are generated independently using four different generative AI models. In order to select suitable models, the first step was to determine which type of model to focus on - generative adversarial networks (GANs) or diffusion models. This decision was necessary because limited resources meant that restrictions had to be defined, which also affected the range of generators analysed.
After thorough research and initial local tests in operation, diffusion models were chosen. There were several reasons for this: one key aspect was their good availability and significantly easier handling in practice. In contrast, many GAN models are only available in the form of GitHub repositories, some of which are outdated or no longer maintained. This led to incompatibilities with current software in the tests, meaning that either outdated execution environments had to be set up or software rollbacks had to be performed in order to run them.
Another decisive selection parameter was practical usability on a graphics card with only 12 GB VRAM. Models that required higher graphics memory requirements by default - and could not be configured to be more resource-efficient through simple adjustments in the script - were rejected. Details on this follow in the next section. In addition, it was assumed that the models would be able to generate images natively in 512 x 512 px format to avoid subsequent cropping or scaling operations.
Furthermore, the models should already have been used in other scientific work to ensure better comparability of the results. In particular, the work of Mallet et al. (2025) was used as a reference, in which several of the generators used in this work were also used. [1]
The following lists all AI models and AI providers that were used to generate the artificial data sets. All generators originate from the *Hugging Face* platform (https://huggingface.co), an open, community-based platform (GitHub for AI) for sharing, versioning and executing pre-trained models, datasets and AI apps. [30]

The generated images are output by the generators as *.png* files by default.

Lykon/dreamshaper-8 *DreamShaper-8* is a finely tuned derivative of *Stable Diffusion 1.5*, which Lykon has released as a versatile text-to-image tool for illustration, anime aesthetics and moderately photorealistic scenes. With around 2 GB of Safetensor weights, it can be loaded directly into Diffusers and delivers coherent results starting at 512 x 512 pixels with just 25 to 30 inference steps; upscaling to 1024 pixels preserves the level of detail. Compared to version 7, LoRA support and style range have been expanded, while the negative embeddings *BadDream* and *UnrealisticDream* provided by the author effectively suppress typical anatomy and noise artefacts. [31] [32]

The model is licensed under the CreativeML Open RAIL-M licence, which allows commercial use in compliance with minimum ethical standards. Due to its balanced combination of stylistic diversity, quality level and low VRAM requirements, DreamShaper-8 is considered a popular *Swiss Army knife* for prompt-based image synthesis in the open source community. [31] [33]

In the following chapters we reference this model as *Dreamshaper*.

PixArt-alpha/PixArt-XL-2-512x512 *PixArt-XL-2-512 x 512* is a diffusion transformer instance from the PixArt Alpha family that replaces the conventional UNet with a purely transformer-based network in latent space. In combination with a fixed T5 text encoder and the VAE introduced by Rombach et al., a single sampling pass is sufficient to generate 512-pixel images; thanks to the scale-invariant architecture, prompts can be upscaled to 1024 px without a separate upscaler. The parameter stock is around 600 million, making the model slightly larger than *Stable Diffusion v1.5*, but significantly leaner than Imagen or DALLE-2. [34]

Despite its moderate size, PixArt-Alpha demonstrates remarkable training economy: the authors report only 25 million image-text pairs and 675 A100 GPU days - about one-tenth of the effort required for SD v1.5 and less than one percent of the resources allocated for the SOTA benchmark RAPHAEL. User studies confirm that the architecture has equivalent or better preference values compared to SDXL 0.9, Stable Diffusion 2, DALLE-2 and DeepFloyd. [34] [35] [36]

Under the CreativeML Open RAIL++-M licence, PixArt-XL-2-512 x 512 can be used commercially, provided that the ethical licence requirements are observed. However, the model card highlights typical diffusion weaknesses - limited photorealism, unreliable hand synthesis and lack of text readability. Nevertheless, the ability to efficiently infer the model on GPUs with approxi-

mately 12 GB VRAM or more using torch.compile or CPU offloading establishes it as a lightweight, transformer-based alternative to UNet-based stable diffusion pipelines. [34]
In the following chapters we reference this model as *PixArt or PixArt Alpha*.

playgroundai/playground-v2-512px-base The *Playground v2512px Base* model is a text-to-image generator based on latent diffusion models and was developed independently by Playground AI. It generates images with a fixed resolution of 512x512 pixels and was designed for research and evaluation purposes. Compared to aesthetically optimised models, the visual output is deliberately kept neutral to enable fair and reproducible evaluation. Technically, the model is based on an architecture that is strongly inspired by Stable Diffusion XL, but uses two text encoders - OpenCLIP-ViT/G and CLIP-ViT/L - to achieve improved semantics and text-image coherence. A key feature of Playground v2 is the publication of intermediate versions (so-called *intermediate checkpoints*), which allow the development of the model to be tracked step by step. This promotes transparency and traceability in model research. The quality of the model was evaluated in a comprehensive usage study involving over 2,600 different text inputs from several thousand people. The images generated by Playground v2 were rated approximately 2.5 times more often as visually appealing and contextually appropriate than those generated by Stable Diffusion XL. For objective evaluation, the MJHQ-30K dataset was introduced, a benchmark with ten different categories, each containing 3,000 examples curated by Midjourney. It enables a differentiated analysis of aesthetic quality and semantic precision using CLIP scores and other metrics. In quantitative benchmarks, *Playground v2 512px Bas*e achieves a FID value of 9.55 and a CLIP score of 32.08 on the COCO14 dataset - values that are significantly higher than those of SDXL 1.0 Refiner, which was rated with an FID of 13.04 and a CLIP score of 32.62. [37]
In terms of licensing, the model is subject to the Playground v2 Community License, which allows both scientific and commercial use, but imposes restrictions once the number of users exceeds one million per month. [38]
In the following chapters we reference this model as *Playground*.

stable-diffusion-v1-5/stable-diffusion-v1-5 *Stable Diffusion v1-5* is a latent text-to-image diffusion model that can generate photorealistic images from arbitrary text inputs. The checkpoint is based on the weights of Stable Diffusion v1-2 and was then fine-tuned with 595,000 training steps at a resolution of 512x512 on the laion-aesthetics v2 5+ dataset, with an additional 10%

of text conditioning deleted to improve classifier-free guidance sampling. The use of the official text encoder CLIP ViT-L/14 enables solid semantic alignment to the prompt. The model corpus was developed by Robin Rombach and Patrick Esser. It is a diffusion model that connects text via cross-attention with a U-Net-based denoiser in latent space, supplemented by a VAE decoder for final image generation. Both pruned EMA-only weights (VRAM-friendly for inference) and full weights for fine-tuning can be used, available as safetensors and usable via the Hugging Face Diffusers library (StableDiffusionPipeline). Licensing is under the CreativeML OpenRAIL-M licence, which allows free use but provides for certain restrictions, such as in cases of misuse or sensitive use. Typical use cases include research into the safe provision of generative models, artistic applications and educational purposes; problematic uses such as discriminatory, violent or copyright-infringing content are marked as not permitted. The model has clear limitations in terms of its scope of use: it does not achieve perfect photorealism, cannot generate legible text, reaches its limits with complex compositions such as multiple objects, and struggles with the realistic generation of faces. In addition, performance is significantly lower for non-English prompts, as training was primarily conducted with English captions. [39]
In the following chapters we reference this model as *SD15 or Stable Diffusion 1.5*.

4.4. Sample size

In this study, a dataset size of 10,000 images was selected for the general experiments and predictive analysis in order to obtain statistically meaningful and robust results. This size represents a practical compromise between empirical significance and available resources. In particular, the available computing resources affect several aspects of the experimental design: the creation time of the deepfakes, the computing effort required to apply steganographic techniques, and the analysis time required for steganography analysis. The scope of the selected data sets is also based on comparable work from current research literature. For example, a similarly large data set was used in the study by Mallet et al. (2024), which supports methodological comparability. [1]
The ALASKA2 dataset, which comprises a total of 80,000 images, was used for the analysis of real image data. A subset of 10,000 images was extracted from this total stock using a specially developed Python script that implements random selection. This reduction was necessary in order to bring the amount of data into a realistic relationship with the computing effort without compromising the representativeness of the data.

When generating artificial image datasets, the target size was defined in advance. 10,000 images were generated per generator, so that an equally large, controlled database was available for each generation mechanism.
Smaller datasets were used to supplement the DCI analysis, which quantifies the consistency between predictions and local sensitivities. These comprised 400 images per evaluation. The choice of this reduced size results from a methodologically justified compromise between the accuracy of the estimated metrics and the associated computational effort. The specific determination of the required sample size was based on statistical considerations, with the confidence interval for proportion values modelled. Assuming an unknown true error rate and a maximum tolerated error of $\pm$ 5 percentage points, it was shown that a minimum sample size of 371 images is necessary to estimate the error rate with 95% confidence within this error margin. For practical reasons, the number was rounded up to 400 images.

4.4.1. Derivation of the minimum sample size for DCI analysis

To estimate the DCI score in steganalysis, the error rate is considered - the proportion of images that the analysis model incorrectly classifies. In the absence of prior knowledge, the worst case $p = 0{,}5$ is conservatively assumed because this value maximises the variance of the binomial distribution. The two-sided confidence level is $95\,\%$, so the standard normal quantile $z_{0{,}975} = 1{,}96$ applies. The accuracy requirement is a maximum allowed half-width of the confidence interval of $E = 0{,}05$ (5 percentage points).

Formula

The sample size estimation is based on the standard formula for estimating the required number of samples when proportions in large populations are considered:

$$n_0 = \frac{z^2 \cdot p(1-p)}{E^2}$$

Inserting the values yields:

$$n_0 = \frac{(1.96)^2 \cdot 0.5(1-0.5)}{0.05^2} = \frac{3.8416 \cdot 0.25}{0.0025} = \frac{0.9604}{0.0025} = 384.16$$

Since the total population size is finite ($N = 10{,}000$), a correction using the *finite population correction (FPC)* is applied. The corrected sample size is calculated as follows:

$$n = \frac{n_0}{1 + \left(\frac{n_0 - 1}{N}\right)} = \frac{385}{1 + \left(\frac{384}{10{,}000}\right)} = \frac{385}{1.0384} \approx 370.8$$

Results

Since a sample size must be an integer, it is always rounded up:

$$\boxed{n_{\text{min}} = 371}.$$

At least 371 randomly selected images are therefore sufficient to estimate the DCI score to an accuracy of $\pm$ 5 percentage points (95% confidence) - even under the pessimistic assumption of an actual error rate of 50%. In the course of the work, this was rounded up to 400.
The authors of the steganography analysis tool used recommend a data set size of at least 10 objects, but this was deemed insufficient in this study due to the scientific environment and the impact on the significance of the results. [28]

4.5. Data collection methods

This section explains how to obtain or generate the data sets described above.

4.5.1. Acquisition of real data set

For the real data set, a shell script for the desired data set - in this case, the uncompressed colour data set in 512x512 format - is downloaded from the website using *wget*. The shell script is then executed. The script automatically downloads the entire *ALASKA v2* image data set *ALASKA_v2_TIFF_512_COLOR* from the URL *alaska.utt.fr*. It creates the target directory, derives the file extension from the data set name, and then iterates from 00001 to 80005, checking before each retrieval whether the file in question already exists locally. If it is missing, it is first downloaded to a temporary directory using *wget* and then moved to the final destination directory; all output is redirected to a log file. After every ten downloads, the script displays a progress message with the time elapsed so far. No pre-compressed version of the data set is downloaded, even though it is available in various quality levels, as these were compressed using different methods than those used in this work.

4.5.2. Generation of artificial data sets

A standardised process was defined for generating the artificial image datasets, which was applied to all models used. Each model was run within a controlled, isolated Python environment, using central libraries such as diffusers and torch. The models were obtained via Hugging Face and used automatically for image synthesis using customised Python scripts. At least 10,000 images were generated per model to provide a sufficient statistical basis for downstream analyses. Particular emphasis was placed on reproducibility and resource-efficient execution, taking GPU capacities into account. To systematically detect and remove potentially unusable image outputs (e.g. completely blacked-out images), custom validation and deletion scripts were implemented. The entire process chain is designed to ensure both scalability and consistency across different model architectures.

4.6. Data processing and quality assurance

This chapter explains the steps involved in preparing the data sets. This includes JPEG compression in three quality levels and the embedding of payloads using two different steganographic methods.

4.6.1. JPEG compression

The first essential processing step involves targeted JPEG compression of the generated image data. For this purpose, a Python script is used to read in the respective raw data sets and then save the images at the selected quality levels. Each raw data set - both those from the various image generators and the real data set - is compressed three times, with quality factors of 75, 90 and 95. The choice of these specific quality levels is based both on their practical relevance and their prevalence in scientific literature. Quality 75 corresponds to the default value of many widely used JPEG compression libraries such as libjpeg and is frequently used in everyday applications. Quality 90 is a common setting in image editing programmes and mobile devices, while quality 95 is often used in professional workflows that require high visual quality with moderate compression. In addition, the ALASKA2 dataset used to train the models provided by the steganography analysis tool also contains images with exactly these quality levels. The chosen compression strategy thus ensures a realistic, differentiated analysis while contributing to comparability and consistency with

established reference datasets.

4.6.2. Embedding the payload using steganography techniques

After JPEG compression, a randomised payload is embedded using two established methods: J-MiPOD and J-UNIWARD. Both methods are among the most widely used approaches in JPEG steganography and are widely recognised in relevant research. The theoretical foundations of these algorithms are described in detail in chapter 2. For implementation, the simulators integrated in the steganography analysis tool are used, which access functional code implementations of the respective embedding methods in the background. These simulators make it possible to automatically embed the payload into the respective image using the desired method and in a specified size. A constant rate of 0.4 bpnzAC is selected as the payload. The characteristic value *bpnzAC* (bits per non-zero AC DCT coefficient) describes the number of embedded bits relative to the total number of non-zero AC DCT coefficients in a JPEG image. Since only these DCT coefficients are relevant for embedding and their number depends heavily on the image content and the selected quality factor, *bpnzAC* provides a comparison value that is independent of the image and its quality. For an image with N_{nzAC} non-zero AC DCT coefficients and an embedded payload of B bits, the result is:

$$\text{bpnz AC} = \frac{B}{N_{\text{nzAC}}}$$

The chosen embedding rate of 0.4 bpnzAC represents a proven compromise it offers sufficient capacity to train steganalysis methods reliably, but does not cause changes that are so noticeable that detection would become trivial. This rate typically corresponds to about 0.1 to 0.2 bits per pixel at quality levels between 75 and 95. It is also used in authoritative benchmark datasets such as ALASKA2 and in numerous comparative studies with methods such as J-UNIWARD, MiPOD, and UERD. This allows the results generated to be easily compared with other work and at the same time ensures a consistent training basis for machine learning methods. [40] [41] [29] [1]
Each data set in each quality level is modified exactly once with J-MiPOD and once with J-UNIWARD, so that a corresponding steganographically modified image set is available for each combination of generator, quality level and embedding method. After completing this step, a structured and consistent data set is available that covers all relevant combinations for the subsequent steganography analysis. After completing the compression and embedding steps, a total of 30

data sets (4 artificial sources x 2 embedding algorithms x 3 quality levels + 1 real data source x 2 embedding algorithms x 3 quality levels) are now available for further analysis.

4.7. Evaluation and experiment design

This subchapter describes the hardware and software used, as well as the tools used. Also experiments and scenarios aswell as metrics are described.

4.7.1. Setup

Hardware

The author's local hardware environment is used for all processing steps. The system is based on an Intel Core i9-10900F with a clock speed of 2.80 GHz and has 32 GB of RAM.
An NVIDIA GeForce RTX 4070 SUPER with 12 GB VRAM is used as the GPU. The operating system is Debian 12. This configuration enables efficient execution of all necessary steps, including image compression, steganographic embedding and model inference.

Software

The software relevant for the creation of this thesis and the execution of the research section is listed below. Software critical to the research is explained in more detail.

Aletheia

Aletheia is a freely available open-source tool for image-based steganography analysis, featuring a modular Python command line interface and an MIT licence. It combines classic steganalysis approaches with modern deep learning methods and enables both the detection of simple LSB manipulations and the analysis of complex embedding schemes such as J-UNIWARD, HILL or SteganoGAN. The range of functions extends from automated quick analyses and classic structural attacks and calibration tests to brute force methods against common tools. Aletheia is therefore aimed at forensic users who require efficient and reproducible results, as well as the scientific community. Aletheia particular emphasis on integrated simulators that can be used to generate synthetic stego data sets for more than twenty common steganography methods. This feature uses either native Python implementations or integrated original implementations in GNU Octave, allowing robust and comparable test collections to be generated without additional configuration

effort. For analysis, Aletheia relies on a combination of pre-trained EfficientNet-B0 models, classic feature-based extractors (such as SRM, GFR, DCTR) and an internal cover-source detection (DCI) method. This DCI module checks whether the model used matches the image source and, in the event of a cover source mismatch, can automatically warn and recommend alternative models. The entire model pipeline is clearly structured. First, integrated simulators generate JPEG stego data sets, for example with the commands *j-mipod-sim* or *j-uniward-sim*. These allow images with either fixed or random payloads to be generated automatically from any cover directories, with the embedding being identical to the original implementation. The data is then split into training, validation and test sets using *split-sets* or *split-sets-dci*. The DCI method extends this structure with additional double stego images, which allow systematic verification of whether a detector can reliably separate under realistic conditions. Two models are trained for each method: an A model that distinguishes between cover and stego, and a B model for separating stego and double stego. Both models use identical training parameters, including Adam optimisation, batch size 32, early stopping in case of stagnation, and targeted data augmentation to increase robustness against JPEG artefacts. Evaluation is performed using *effnetb0-predict* for standard cases or *effnetb0-dci-predict* for CSM-critical scenarios. The latter combines the classification probabilities from the A and B models and indicates possible mismatches as soon as the estimated model confidence falls below approximately 0.7. All common variants are already trained and available in Aletheia's Model Zoo. Under ideal conditions on Alaska2 data, the following separation accuracies result for the methods relevant to this thesis: For J-MiPOD, the A model achieves an accuracy of 0.837, the B model 0.670; for J-UNIWARD, the corresponding values are 0.759 and 0.698, respectively. These values serve as upper limits for the expected detection performance with ideally matched sources and are adjusted accordingly by the DCI module in real-world scenarios. [28]

The development of Aletheia is led by Daniel Lerch-Hostalot and David Megías at the Universitat Oberta de Catalunya (Barcelona). Lerch-Hostalot has many years of research experience in the field of steganography and maintains StegoLab, a comprehensive collection of free tools. Megías leads a research group at the IN3 Institute and is co-author of numerous studies on cover source mismatch and steganographic security. [42], [43]

Aletheia is used as the central technical component in this thesis. It serves both to generate steganographically altered images using the integrated simulators and to apply pre-trained models to specially generated data sets. In addition, it is used to carry out our own training runs in order to evaluate experimental models under specific parameters and source conditions. The tool is

installed according to the recommended method via *pip* so that all required packages and software are installed automatically.

Hugging Face CLI

To work with Hugging Face models, users must first install the command line interface (huggingface-cli). In this environment, version $huggingface_hub == 0.33.2$ is used. After successful installation, authentication takes place via a personal access token, which is generated directly via the user account on Hugging Face and then stored via the CLI. This enables access to private models, the downloading of publicly available resources and integration into local workflows.

4.7.2. Scenarios

In an initial application scenario, Aletheia is used to apply the pre-trained A models based on the ALASKA2 dataset to all available data. All raw data sets in all quality levels are taken into account, as are all steganographically modified versions of the same. The aim of this scenario is to investigate the generalisation ability of the models. Through the systematic analysis of all variants, mean values, total error rates and other statistical indicators can be calculated, providing detailed insights into the recognition performance and model robustness. In the second scenario, the pre-trained B model is used for a DCI analysis. Here, a smaller, specifically selected subset of all data sets is used to check the extent to which the pre-trained models are applicable to the tested data sources. The confidence values determined in this process provide information about the reliability of the classification outputs and are particularly important in the context of cover-source mismatch. They show whether there is a potential mismatch between the model and the data source and whether the model can deliver reliable results in the respective scenario. A third scenario aims to create and apply specially trained models. Two separate models are trained for each generator - one for J-MiPOD and one for J-UNIWARD. All available quality levels of a generator are combined into a training data set; a defined part of the data set is reserved for validation and scoring purposes. This results in a total of eight separate models. These are then used in the same way as in the first scenario to compare the new results with the original Aletheia pre-training models. This comparison allows conclusions to be drawn about the model adaptation to specific generators and about possible improvements in the separation accuracy.

4.7.3. Metrics

Five key metrics are used to evaluate the steganography performance, taking into account cover-source mismatch (CSM): the *average prediction score*, the *DCI score*, the *total error rate*, the *intrinsic difficulty* and the *inconsistency*. These are based on a statistical hypothesis test that distinguishes between two classes: H_0 (cover image) and H_1 (stego image).

Prediction Score

The predict value is the raw output of the steganography analysis model for exactly one object/image. The last network layer contains a single sigmoid activation, the result of which lies between 0 and 1. This number is interpreted as the estimated probability that the image contains stego data. The closer the value is to 1, the more confident the model is that there is an embedding in the image; the closer it is to 0, the more it suspects an unchanged cover. Values around 0.5 mean that the features present in the image cannot be clearly assigned to either class by the network - in such a situation, it would be practically uncertain and would only guess if a hard decision had to be made.
The value therefore says nothing about how often the network is fundamentally correct - that would be a hit rate for which ground truth labels or a DCI estimate would be needed. Nor is it a threshold value that Aletheia already sets. If you want to force a yes/no decision, you have to choose a limit yourself, for example 0.5 or - for greater scepticism - a higher threshold such as 0.7, depending on whether you prefer to minimise false alarms or oversights. In summary, the predict value is nothing more than the model's immediate, probabilistic judgement of a single image, not a global quality metric.

DCI Score (Detection of Classifier Inconsistencies)

The so-called detection-of-classifier-inconsistencies value (DCI) is a prognostic indicator of how reliably a previously trained steganography classifier A works on an unlabelled image set. Each source image x_i is first classified by A into the classes $\{0, 1\} = \{\text{Cover}, \text{Stego}\}$. Then, a second payload is embedded in the same image using a known simulator, resulting in the variant x_i^{double}; A second network B works on this variant, distinguishing between $\{\text{Stego}, \text{Double-Stego}\}$. If the two classifications are logically consistent with each other - formally, for example,

$$\big[y_A(x_i) = 0 \wedge y_B(x_i^{\text{double}}) = 1\big] \ \vee \ \big[y_A(x_i) = 1 \wedge y_B(x_i^{\text{double}}) = 1\big],$$

the image pair is considered *consistent*; otherwise, there is an inconsistency (*INC*). Let n be the number of all pairs checked and K the number of consistencies. The point estimator of the DCI score is

$$\hat{p} = \frac{K}{n} = 1 - \frac{n-K}{n}. \tag{4.1}$$

Each pair decision corresponds to a Bernoulli experiment $X_i \sim \mathrm{Bernoulli}(p)$, so that for the sum $K = \sum_{i=1}^{n} X_i$ applies the following

$$K \sim \mathrm{Binomial}(n, p).$$

This allows a $95\,\%$ confidence interval to be determined using the so-called Wilson interval:

$$\hat{p}_{\mathrm{Wilson}} = \frac{\hat{p} + \frac{z^2}{2n} \pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}, \qquad z = 1{,}96. \tag{4.2}$$

If the estimator is to be published with a maximum specified half-width E, the inverted normal approximation yields the minimum sample size

$$n \geq \frac{z^2 p(1-p)}{E^2}. \tag{4.3}$$

A high DCI value, for example $\hat{p} \geq 0.80$, implies under the above assumptions that the classifier A is highly likely to exhibit the same reliability that it had in its training domain. A significantly lower value, such as $\hat{p} \leq 0.65$, on the other hand, indicates a pronounced cover-source mismatch or an unknown steganography technique; in this case, the classifier must be readjusted or retrained. However, these thresholds are freely definable and are not specifically defined in works and must be individually adapted to the requirements of the steganography analysis tool. Basically, this means that the higher the value, the higher the reliability.
The indicator is used in this work to determine the reliability of the models used, provided by the steganography analysis tool and trained on a real data set, on artificial data sets.

Total Error Rate

The *Total Error Rate* (TER) describes the average error rate of a binary classifier assuming equal a priori probabilities for both classes. For an input image X, which is assigned a score via a scoring function $\lambda(X)$, a threshold value $\tau \in \mathbb{R}$ determines whether the image is classified as cover or stego:

$$\delta(X) = \begin{cases} H_0 & \text{if } \lambda(X) \leq \tau \\ H_1 & \text{if } \lambda(X) > \tau \end{cases}$$

This results in two types of errors:

- *False Positive Rate (FPR)*: $\mathbb{P}(\lambda(X) > \tau \mid H_0)$
- *False Negative Rate (FNR)*: $\mathbb{P}(\lambda(X) \leq \tau \mid H_1)$

The *Total Error Rate* is calculated as the average of both error types at the optimal threshold value:

$$P_E = \min_{\tau \in \mathbb{R}} \frac{1}{2} \left(\mathbb{P}(\lambda(X) > \tau \mid H_0) + \mathbb{P}(\lambda(X) \leq \tau \mid H_1) \right)$$

[1]

Intrinsic Difficulty

The *intrinsic difficulty* characterises the minimum error probability when the training and test data match perfectly, i.e. when a classifier is trained on images from a source S_A and tested on the same source. It is formally defined by:

$$P_E^{(S_A)} = \min_{\tau \in \mathbb{R}} \frac{1}{2} \left(\mathbb{P}^{(S_A)}(\lambda(X) > \tau \mid H_0; S_A) + \mathbb{P}^{(S_A)}(\lambda(X) \leq \tau \mid H_1; S_A) \right)$$

This size serves as a reference value for stegographic recognition performance without source mismatch.

[1]

Inconsistency

The *inconsistency* describes the deterioration in recognition performance in the event of a cover source mismatch. A classifier is trained on images from one source S_A and tested on images from another source S_B. The resulting error probability is calculated in the same way as the total error rate:

$$P_E^{(S_A \to S_B)} = \min_{\tau \in \mathbb{R}} \frac{1}{2} \left(\mathbb{P}^{(S_A)}(\lambda(X) > \tau \mid H_0; S_B) + \mathbb{P}^{(S_A)}(\lambda(X) \leq \tau \mid H_1; S_B) \right)$$

A comparison of this value with the *intrinsic difficulty* of the target source S_B allows conclusions to be drawn about the robustness of the classifier with respect to statistical source shifts. Large deviations indicate strong inconsistencies due to CSM.

[1]

4.7.4. Instruments for data collection in experiments

For data collection, the output of the command line analyses (Stdout) is written directly to separate files. To ensure better further processing and compatibility with existing evaluation routines, the data is stored in CSV format.
Each analysis generates its own output document, which is then cleaned of unnecessary meta-data. Initial evaluations are performed in each of these files, including the calculation of the total error rate at a minimum threshold value and the determination of average values. The resulting key figures are then centrally consolidated in a common results table.

4.8. Limitations

The scope of this work was deliberately limited in order to ensure a clear structure and realistic feasibility. Only publications in German and English were considered during the literature search in order to ensure a linguistically accessible and high-quality source base. In the practical part, the work is limited to classical steganographic methods in the "JPEG domain", specifically to the algorithms "J-MiPOD" and "J-UNIWARD". Steganographic methods based on artificial intelligence and methods in the "spatial image domain" were explicitly excluded. Only "diffusion models" were used to generate the test images, while "GANs" were deliberately not taken into account. Pre-trained EfficientNet B0 models from third parties were used for the steganalysis. All models used - both for steganography and steganalysis - had to be able to run with a maximum GPU memory of 12 gigabytes of VRAM to ensure practical reproducibility. The payload rate when embedding the hidden data was kept constant across all experiments at 0.4 bpnzAC. This work only analysis color images in 512x512 format.

5. Approach

5.1. Implementation

The following chapters describe the procurement and generation of data sets and their processing (e.g., compression, data type conversion). In addition, the process of injecting the payload and the subsequent steganalysis using Aletheia's pre-trained models and own fine-tuned models are explained.

5.1.1. Data generation

This chapter explains in detail the generation, acquisition, preparation and processing of image data for the subsequent scientific investigations. In addition to image generation, particular attention is paid to image compression, which is a central component of data preparation.

Generation of artificial images

In the practical implementation phase, separate Python scripts were developed for each text-to-image model based on the *Start here* scripts provided by Hugging Face. These were adapted so that they work without prompt input and instead generate random images based on the models' training data. The goal was to automatically generate 10,000 images per model without manual interaction. The hardware used had a single GPU with 12 GB VRAM, which meant that the generation and subsequent analysis steps had to be performed sequentially. Each model was run in isolation to avoid memory conflicts. The core components used were the libraries *diffusers* and *torch* from *DiffusionPipeline*. The generated images were saved in PNG format, with a counter in the main loop controlling the number and naming of the images.

```
1    from diffusers import DiffusionPipeline
2
3    model_id = "Lykon/dreamshaper-8"
```

```
    pipeline = DiffusionPipeline.from_pretrained(model_id)

    prompt = ""
    pipeline = pipeline.to("cuda")

    for i in range(1, 10_001):
        # Generate the image
        image = pipeline(prompt).images[0]

        # Save the image with a count-based filename
        image.save(f"image_{i}.png")

        print(f"Saved image_{i}.png")
```

Listing 5.1: Sample Python script to generate 10 000 images with Dreamshaper-8

However, a problem arose when using the PixArt Alpha model. Although documentation and literature research indicated that it should be possible to run the model on a graphics card with 12 GB VRAM, initial tests showed that the model attempted to allocate significantly more graphics memory than was available. Due to this limitation, an alternative script had to be used, which is specially optimised for reduced memory requirements (8 GB VRAM). The developers of the model officially provide such a script, which enables a significantly more resource-efficient execution.
The reduction in GPU memory consumption achieved by the script is based on five interrelated measures:

- *(1) 8-bit quantisation* loads the entire text encoder with `load_in_8bit=True` in 8-bit weights, reducing its parameter and activation memory by at least half compared to 16- or 32-bit representations.
- *(2) Automatic offloading* uses `device_map=balanced` to distribute less latency-critical model parts (tokeniser, text encoder, scheduler) to the host RAM, while only UNet and VAE remain permanently in the GPU.
- *(3) One-time prompt encoding and subsequent release* calculates the text embeddings in advance, then explicitly deletes the text encoder (`del`) and empties the CUDA cache (`torch.cuda.empty_cache()`), so that this large part of the model no longer occupies VRAM.
- *(4) Half-precision weights* (`torch_dtype=torch.float16`) additionally halve the size of

all parameters and intermediate tensors remaining on the GPU.

- *(5) Memory-saving inference loop* disables gradient tracking with `with torch.no_grad()`, initially generates the outputs as latent (`output_type=latent`) and performs an explicit garbage collection and cache flush after each image, immediately freeing temporary tensors.

The combination of these measures typically reduces the maximum VRAM peak from well over 20 GB to around 6-8 GB without any noticeable loss of image quality and speed. After initial successful tests with this script, it was adapted to individual requirements. The original script and the script used can be found in the appendix.

In addition to this problem, others arose during the generation process. During generation, it was discovered that individual models - in particular Stable Diffusion 1.5 and Dreamshaper - occasionally generated so-called *safe* images, which were completely blacked out by NSFW (Not Safe For Work) security filters. Since no prompts were used for image generation, the models themselves are responsible for the content. This can result in content being produced that is classified as inappropriate by these built-in filters in some form. This may be because violence, sexuality or other problematic topics are depicted. These images were easily identified by their file size (less than 1 KB). To automatically clean up these outputs, we developed our own Python scripts that analyse the target directory, identify PNG files with insufficient file size, and delete them. Another solution would have been to adapt the code. However, we decided against this for ethical reasons. The following Python script deletes all `.png` files in a directory whose size is less than 1 kB.

```
import os

folder_path = os.getcwd()
min_size_bytes = 1024

deleted_count = 0

for filename in os.listdir(folder_path):
    if filename.lower().endswith(".png"):
        file_path = os.path.join(folder_path, filename)
        try:
            file_size = os.path.getsize(file_path)
            if file_size < min_size_bytes:
                os.remove(file_path)
                print(f"Deleted: {filename} ({file_size} bytes)")
```

```
16                 deleted_count += 1
17         except Exception as e:
18             print(f"Error checking file {filename}: {e}")
19
20 print(f"Total deleted images: {deleted_count}")
```

Listing 5.2: Python script for removing blacked-out images

The missing images were then regenerated, with a safety margin added in each case to avoid time-consuming regeneration. In later models, image generation was dimensioned from the outset to allow for partial loss due to safety filters. The final data sets therefore exceeded the target size of 10,000 images in some cases; a restriction to exactly 10,000 images was only imposed later in the evaluation phase. The complete cleaning and normalisation of the data sets is explained in more detail in the chapter on result evaluation. The generation time averaged approximately 6 seconds with 50 iterations per image.

Acquisition of the real data set

This section explains in more detail how the real data set was obtained. Information about the data set itself is provided in the chapter 4. As already explained in the methodology section, the Shell script was first downloaded from the official website using *wget*, which was then used to download the data set in 512x512 format, in colour and uncompressed.
Command for downloading the script:

```
wget https://alaska.utt.fr/ALASKA\_v2\_TIFF\_512\_COLOR.sh
```

In summary, the script uses *wget* to download the images from their server in a controlled manner with rate limiting. It always checks whether an image already exists locally. Information about the downloaded images and the elapsed time is output in batches of 10 images to show the progress. The script is included in the appendix. The objects in the dataset were delivered in .tiff format.
As mentioned in the methodology section, the dataset consists of 80k files, which must be reduced in size for this project. For this purpose, a separate Python script was prepared, which copies a specified number of objects from a source directory to a target directory. This is done randomly in order to maintain the heterogeneity of the data set as much as possible. Before a randomly selected object is copied, the script always checks whether it has already been selected and already exists in the target directory. This prevents duplicates in the subset. Another feature of

the script is that specific file types can be specified. This is useful if there are several different file types in the source directories, but only a specific type is to be processed. The script can be found here: A.4.1
The generated subset now consists of 10k objects; a buffer is not required in this case.

5.1.2. Image compression

This chapter discusses the compression of data sets. The focus is on the scripts and quality factors used. JPEG compression itself and the underlying processes and algorithms are not explained. Information on this can be found here: 2.2 Compression is an essential part of data preparation, as this work focuses on the JPEG domain in steganography. The difference between the spatial and JPEG domains is defined in the chapter 2.1.1. Two Python scripts were used for compression. However, these differ only in that one script compresses .png files and the other compresses .tif files. Regardless of the file formats, the images are always in unprocessed raw format (cropping to 512x512 is not considered processing here), meaning that the full image information is still contained. The models used to generate the artificial images provided .png files, while Alasaka 2 provided the data set with .tif files. Using the scripts, all data sets were compressed three times at quality levels 75, 90 and 95 and stored for further processing.

Script explaination of PNG to JPEG

The script (A.3.1) imports four modules: *argparser* for elegantly parsing command line arguments; *pathlib.Path*, which simplifies platform-independent path manipulation; *PIL.Image* from the Pillow library to open, edit and save image files; and *sys* to terminate the programme with specific return codes. The central task is performed by *convert_png_to_jpg*. This function uses *Path.glob(*.png)* to collect all .png files in the specified source directory. If none exist, it issues a warning and returns immediately. Otherwise, it creates the target directory if necessary. For each .png file found, it opens the image, converts it to the RGB colour space (JPEG does not support transparency) and writes it to the destination folder with the same root name and the extension *.jpg*. The call *save(..., JPEG, quality=quality, optimize=True)* passes the RGB pixels to the library *libjpeg*, which is responsible for the actual JPEG compression; the quality value determines the strength of the compression, and *optimize=True* ensures slightly smaller files. Next, *self_check* performs a plausibility check: it builds sets of file stems for all source PNGs and all generated JPEGs (which

must not be empty). If a JPEG is missing for a PNG or vice versa, it reports this and returns *False*; if everything is correct, it reports success and returns *True*. The main function uses *argparse* to set three parameters on the command line: input folder, output folder and, optionally, the desired JPEG quality (default 85). If it determines that the input path is not a directory, it immediately aborts with an error message. It then calls the conversion, followed by the self-test, and exits the programme with a return code of 0 if successful or 1 if there are problems - a behaviour that makes the script easy to integrate into automated workflows.

The second script (A.3.2) serves the same purpose as the first, namely converting TIFs to JPEGs and then checking whether each source file has been successfully converted; however, the main difference lies in the file type. While the original programme only uses files with the extension *.png*, the new script targets TIF files and accepts both upper and lower case (tif and tiff in all variations). To cover this flexibility, it defines its own helper function *list_tif_files*, which queries each relevant extension individually instead of using a simple *glob(*.png)* as before. Because TIFF files often contain mixed upper and lower case letters, the program also takes care to search each of the four possible suffixes separately. During self-checking, it therefore compares the roots of the TIFF files found with this helper function with the roots of the generated JPEGs; in the PNG script, a direct *glob(*.png)* was sufficient. In addition, the new script validates the quality value passed and aborts if it is outside the range one to ninety-five, while the PNG counterpart simply sets a default value and does not perform an explicit range check. The rest of the logic remains the same: images are opened with Pillow, converted to RGB format, then compressed using the libjpeg routine and written to the target folder with the same base name and the extension *.jpg*; a self-test then determines whether the process was successful and which return code is sent to the operating system. The script is otherwise identical to the previous script and can be viewed in the appendix.

In summary, the four artificial data sets and the real data set are compressed and stored in three quality levels. A total of 15 ynthetic compressed data sets are now available (5 sources x 3 quality levels).

5.1.3. Steganography

This chapter explains how to use steganographic methods. It also explains how to prepare the environment and install Aletheia.

Preparing the environment

Before Aletheia can be used, a *pip* environment must be configured. This helps to install and manage specific Python package versions without affecting other projects. The tool *pip* is a package manager in Python that installs/manages Python modules and packages. Pip environments offer the advantage of working without administrator rights.
Command:

```
python -m venv /path/to/new/virtual/environment
```

After preparing the environment, Aletheia was installed according to the installation instructions.
Command:

```
pip3 install git+https://github.com/daniellerch/aletheia
```

The command clones the GitHub repository and automatically installs all necessary packages. Once the installation steps were complete, Aletheia was ready to use.

Embedding the payload

Once all data sets have been compressed, the cover images are manipulated using steganographic methods or provided with secret data, known as secrets, as carrier objects. Specifically, the embedding algorithms *J-MiPOD* and *J-UNIWARD* are used and randomised payloads of fixed 0.4 bpnzAC (bits per non-zero AC DCT coefficient) are packed unencrypted into the covers. More detailed information on how these work and their properties for hiding secrets is provided in chapter 2.
Ready-made integrated simulators in Aletheia are used to embed the payloads using both methods. These simulators were developed partly by the developers themselves and partly by external sources. Due to different licences, they are downloaded/uploaded separately after consent has been given and the licence requirements have been accepted. The third-party simulators are mostly Matlab or Octave implementations. [28] For this work, the integrated simulators *j-uniward-color-sim* and *j-mipod-color-sim* are used, as images are processed in colour.
When first called, the main script checks whether the relevant Octave programme is already in the local directory `$HOME/.aletheia/external/octave`. If it is missing, Aletheia loads the file from the supplementary repository *aletheia-external-resources*, displays the corresponding licence and then executes the code as a separate Octave process.

The command *j-mipod-color-sim* uses the routine `J_MiPOD_COLOR.m`, an unmodified copy of the reference source code by Rémi Cogranne *et al.*. The header of the script explicitly refers to the works *Steganography by Minimising Statistical Detectability: The Cases of JPEG and Colour Images* [44] and *Content-Adaptive Steganography by Minimising Statistical Detectability* [45]. The source code can be found at: `https://github.com/daniellerch/aletheia-external-resources/blob/main/octave/code/J_MIPOD_COLOR.m`.

The command *j-uniward-color-sim* is based on `J_UNIWARD_COLOR.m` from the DDE Lab package by Vojtch Holub and Jessica Fridrich, which implements the colour variant of the JPEG UNIWARD method. The file cites the paper *Universal distortion function for steganography in an arbitrary domain* [40] as its scientific source. In Aletheia the code can be found at: `[https://github.com/daniellerch/aletheia-external-resources/blob/main/octave/code/J_UNIWARD_COLOR.m]`.

Both simulators are fed directly from the original implementations of research groups; Aletheia only handles automatic reloading, licence confirmation and command line encapsulation to enable reproducible embedding experiments.

With the help of the simulators, each data set is processed twice (once per embedding method) and randomised secrets are embedded in a fixed payload area of 0.4 bpnzAC. Specifically, this means that three data sets (Q75, Q90, Q95) from each model are processed twice. This results in a total of 15 data sets per embedding procedure.

Commands:

```
./aletheia j-uniward-color-sim /path/from/sourcedir/ 0.4 /path/from/
    destdir/

./aletheia j-mipod-color-sim /path/from/sourcedir/ 0.4 /path/from/destdir
    /
```

The simulators use the CPU, and given the size of the data sets, the process took several hours for J-MiPOD-Color-Sim and over a day for J-UNIWARD-Color-Sim per data set source. This is partly due to the way these procedures work and how they are implemented in terms of code (see chapter 2). The performance of the CPU also plays a significant role, of course. To make the embedding process as efficient as possible, a Bash script was created that executes all commands for all data sets sequentially. The script is not included here for data protection reasons, as it contains the actual paths. It is a sequence of the example commands from above. After the embedding is complete, finished data sets with a fixed payload ratio for each model and the real

image source in each quality level after compression are available for further analysis. The next step is to continue with the steganography analysis.

5.1.4. Steganalysis

This section explains the analysis of the previously discussed data sets using the EfficientNet B0 models integrated into Aletheia. First, we explain how the models were trained, how the models work for steganalysis, and then how the process flow was defined for application in this work.

Aletheia pretrained EfficientNet-B0 models

EfficientNet models form a family of CNNs based on a base model called EfficientNet-B0, which was developed specifically using neural architecture search. All other variants (B1 to B7) are created by applying the compound scaling method with increasing values of ϕ. Each model in the family is thus optimally tailored to a specific resource framework. The scaling relations are based on constant factors determined in a grid search procedure for the initial model and ensure that scaling remains efficient with each increase in ϕ. The EfficientNet-B0 model itself is a lightweight, mobile network consisting of so-called MBConv blocks (Mobile Inverted Bottleneck Convolutions), supplemented by squeeze-and-excitation modules. It uses, for example, 5.3 million parameters and already achieves 76.3% top-1 accuracy on ImageNet at only 0.39 billion FLOPS. This makes it an efficient starting point for further scaling with higher accuracy as computing budget increases, without losing the balance between representativeness and efficiency. [46] Aletheia offers pre-trained models based on EfficientNet-B0 models for steganography analysis using CNNs. For today's state-of-the-art models, the project relies almost exclusively on EfficientNet-B0 networks. The starting point is a large collection of unprocessed cover images from the ALASKA2 database. For each steganography method to be detected (such as Steghide, F5/nsF5, J-UNIWARD, HILL, etc.), Aletheia uses the integrated simulators to generate suitable stego variants; The embedding rate is randomly selected in the range 0.05-0.50 bpp or bpnzAC to make the network more robust against variable payloads. The cover and stego material are then used to form three disjoint subsets for training, validation and testing using the split-sets command (typically 1,000 images each for validation and testing, with the remainder for training). For detection under cover-source mismatch conditions, Aletheia also creates double images with double embedding and separates the data into A and B subsets using split-sets-dci. [28]

Prediction analysis using a pre-trained model

This step involves analysing the data sets using Aletheia's pre-trained models and making predictions for each image in a data set to determine whether it is a cover or stego.
Specifically, the models *effnetb0-A-alaska2-juniw.h5* and *effnetb0-A-alaska2-jmipod.h5* are used. The aim is to gain experience on how these models perform, even though not all data sets are the same or similar to their training data. The results of these analyses can then be used to calculate important metrics such as inconsistency and intrinsic difficulty, as well as error rates. We have explicitly decided not to perform DCI tests first in order to obtain feedback from the tool in advance on the trust value and the applicability of the models for the test data sets.
This analysis is performed in the next step. This provides the *bare* performance as it would be if the models were used on an unknown data set for analysis in a real scenario. For this initial evaluation, each data set - all compressed data sets with and without payload - is analysed with one of the two models used, or the purely compressed data sets are analysed with both. The results are written to a CSV file so that the data can be easily evaluated or statistical analyses can be performed. By default, the values are only output directly in the CLI as *Stdout*. For the evaluation, commands are always issued with the same structure. The command consists of calling the tool, selecting the analysis method (in our case effnetb0-predict), the data set to be analysed with its path, the model for the analysis, the device (CPU or GPU - in our case always GPU for performance reasons) and the target file to which the output is written using output redirection.
Commands:

```
./aletheia.py effnetb0-predict /path/to/JUNIWARD/dataset/ \
aletheia-models/effnetb0-A-alaska2-juniw.h5 0 > /path/to/csv/result.csv

./aletheia.py effnetb0-predict /path/to/JMIPOD/dataset/ \
aletheia-models/effnetb0-A-alaska2-jmipod.h5 0 > /path/to/csv/result.csv
```

The CSV files generated were then subjected to post-processing. By redirecting the console output, additional output that was not relevant for data evaluation was written to the files alongside the actual results. These superfluous entries were completely removed. As explained in the previous chapters, more data records were generated than necessary for security reasons. To create a uniform basis for evaluation, all entries above the ten thousandth data record were deleted. Since the objects in all CSV files had the same order and the same deletion method was applied to each file, the consistency of the data records was maintained. For the subsequent analysis, the result

values were standardised to three decimal places to ensure better comparability. After completing these steps, cleaned CSV files were available for all data sets, which were used to calculate the metrics and perform further statistical analyses, as described in a separate subchapter.

DCI analysis using pre-trained models

This chapter explains DCI analysis using pre-trained models. The DCI value indicates how reliable the model is for a given data set. The closer the value is to 1, the more reliable the model and the resulting prediction values are in the analysis. For example, a DCI value of 0.75 means that the model is correct with a probability of 75%. The threshold value is determined individually depending on the scenario and other influencing factors.

Process description In the first step, additional data sets are generated from the existing ones. Specifically, 400 images are randomly selected from each data set and stored in a new directory. The same Python script that was used in the chapter 5.1.1 is used to create a subset of the 80,000 images in the real data set with randomly selected images. Due to the universal applicability of the script, 400 JPEG files were randomly selected for this application.
The DCI values are then determined using Aletheia's B models, specifically *effentb0-B-alaska2-jmipod.h5* and *effnetb0-B-juniw.h5*. Unlike prediction analysis, DCI analysis only examines data sets with embedded payloads.
The following commands are used for this purpose:

```
./aletheia.py dci j-mipod-color-sim path/to/dataset/ 0

./aletheia.py dci j-uniward-color-sim path/to/dataset/ 0
```

The respective command consists of the program to be executed (*dci* for DCI analysis), a steganography simulator (used as in chapter 5.1.3 for embedding payloads), the path to the stego data set (data set A) and the analysis device. Several steps are processed in the background.
A temporary directory is created in which the double stego images (data set B) are stored. Then, using the simulators with a fixed payload size of 0.4 bpnzAC, a randomised payload is embedded into the images of the specified data set. Since these images have already been steganographically modified, so-called double stego images are created. After preparing the data sets, an analysis is performed using the B model of the two data sets. The selection of the model is defined in the code and, like the payload, does not need to be specified. If the model is suitable,

an additional embedded J-MiPOD message must increase the stego probability or at least keep it constant (pB >= pA).. For each image pair comparison, Aletheia notes in the background whether this condition is met. Violations are marked as inc (inconsistent). After the analysis is complete, an average value is output, which represents the reliability, and the double Stego data set is automatically deleted. The individual steps and their implementation can be found in detail in the source code. [28] To automate the evaluation, a Bash script was created which executes the individual CLI commands sequentially. The results are written to text files for persistence of the outputs and then centrally merged and evaluated.

Fine-tuning of EffnetB0 models and re-analysis of artificial data sets

This subchapter describes the training process for proprietary models based on artificially generated data sets. The aim is to improve the recognition rates of steganography analysis using an atomistic approach. The atomistic approach involves training several specialised models - in this case, two per artificial data set. Strictly speaking, this is not a complete training process, but rather a fine-tuning of pre-trained models. Specifically, EfficientNet-B0 models provided by Aletheia are used and adapted to the specific characteristics of our own data sets. This contrasts with the holistic approach, which is also used in similar scenarios. This would mean that all artificially generated data sets are merged into a comprehensive overall data set and a single model is then optimised. Strictly speaking, this work pursues a combination of both approaches: Although there are a total of six data sets per generator due to the different compression levels, only two models are trained per generator. The approach could therefore be interpreted as partially atomistic with holistic elements. A precise categorisation therefore remains open to discussion. Before the actual fine-tuning, the training data sets must be prepared accordingly. As already mentioned in the previous sections, all compression levels for each steganography method and generator are combined in a common data set. Since the file names within the individual compression levels are identical and only differ in the directory paths, renaming is necessary to avoid overwriting. For this purpose, a separate script was developed that reads the files, adds individual prefixes and moves the renamed files to a defined target directory.
The script can be found here: A.5.1
After the Stego and cover data sets had been completely merged, Aletheia was used to prepare the data structurally for the machine learning module. This module requires a specific directory structure, which is why the data first had to be divided into training, validation and test sets. The

split-sets command was used for this purpose, which accepts the path to both the cover and Stego data sets and then divides the files randomly in a balanced ratio. In addition, parameters can be used to specify the size of the validation and test sets. Since the original data sets each comprise three different quality levels, a size of 3000 objects was defined for the validation and test sets for each quality level.
Command:

```
./aletheia.py split-sets ../Dreamshaper/Komprimiert/all/ ../Dreamshaper/
   Komprimiert/juniward/all/ ./Models/Dreamshaper/juniward/ 3000 3000 0
```

Once the training sets had been prepared and divided accordingly, fine-tuning of the models could begin. To do this, the following command was executed for each generator and each steganography method. This starts the training of an EfficientNet B0 classifier in Aletheia as a so-called A-model, which distinguishes between cover and stego images. The command uses the specified directories for training and validation, whereby cover and stego must be stored in separate folders. The resulting model is saved under a unique name in the form *A-generator-method*, runs on GPU 0, uses early stopping with a patience of 100 (= 100000 x batch size) and trains with a batch size of 14. The batch size defines the number of training examples that are calculated forwards and backwards together by the network in an optimisation step.

```
./aletheia.py effnetb0 ./Models/Dreamshaper/jmipod/train/cover/ ./Models/
   Dreamshaper/jmipod/train/stego/ Models/Dreamshaper/jmipod/valid/cover/
   Models/Dreamshaper/jmipod/valid/stego/ A-Dreamshaper-jmipod 0 100 14
```

The batch size was determined empirically: the largest batch size that could still be processed by the GPU with 12 GB VRAM was selected in each case. Although an early stopping patience of 100 was defined, fine-tuning was manually terminated after 100 epochs for pragmatic reasons. Longer training would not have yielded any additional scientific insights due to limited time resources. In most cases, the models achieved validation accuracies of over 0.9 very early on, indicating that they were already performing at a high level. The same procedure was therefore followed for all training runs: Aletheia automatically evaluated the interim results and saved the current best model under the name '...-best.h5'. After the 100 epochs had elapsed, this *best model* was retained for further analysis.
In summary, a Python script was used to prepare three data sets for each generator:

- a combined set of real images of all quality levels,
- an artificial data set for J-MiPOD,

- and an artificial data set for J-UNIWARD (each with all quality levels).

These were then divided into training, validation and test sets in Aletheia, and the fine-tuning models were trained based on these.

After fine-tuning was completed, the models generated were used to analyse the data sets, following the same procedure as for the original Alaska2-based Aletheia model. All quality levels of the artificially generated, steganographically manipulated data sets, as well as all uninfluenced data sets, were evaluated with each of the trained models. The resulting analysis values were again saved in CSV files, cleaned of metadata and then used for further evaluation and calculation of the metrics.

5.2. Data analysis

The following chapters explain how to evaluate the results of the analyses and how to calculate the defined metrics.

5.2.1. Data analysis of prediction results

Calculation of average values

The first metric is the mean value of the prediction values. This step is performed separately for the results of the J-MiPOD and J-UNIWARD methods. First, all CSV files containing analysis results for data sets with embedded payloads are cleaned of irrelevant output. This output comes from the standard output stream (stdout) and was written to the files by mistake. The number of lines to be analysed is then limited to 10,000 entries to ensure a consistent data set size. In the next step, the arithmetic mean of the prediction values for each file is determined. To do this, the CSV files are opened in an office suite and the integrated table function `AVERAGE` is used. The calculated mean values are then documented centrally in a consolidated `.xlsx` file. This procedure is applied to a total of 30 data sets. The result is a structured table in which all mean values of the prediction analyses for data sets with embedded payload are clearly displayed.

Formula in office suite:

```
=AVERAGE(B1:B10000)
```

Calculation of the Total Error Rate (TER)

The total error rate is derived from the values for the false negative rate (FNR) and the false positive rate (FPR). For each procedure and each quality setting of an image generator, the respective TER value is calculated as the arithmetic mean of FNR and FPR, provided that the threshold value is minimal. To calculate the total error rates, a Python script (A.6.1) is created, which is used to find the optimal threshold value and then calculate the TER. The Python script is used to automatically determine the threshold value at which the total error rate is minimal from any CSV file with classifier scores and binary labels.

When called, it first reads the columns *score* and *label*, checks their existence and ensures that the labels only contain the values 0 (cover) or 1 (stego).

It then sorts the data according to the score, calculates the false negative rate and the false positive rate simultaneously for each possible threshold using cumulative sums, forms the respective TER from these and finally determines the exact score at which this value is smallest.

Finally, the programme outputs both the optimal threshold value τ and the associated minimum error rate, making it particularly suitable for objective threshold optimisation in steganography experiments.

Before the Python script can be executed, the corresponding CSV files must be prepared accordingly to avoid errors and ensure consistency. To do this, a CSV file containing only scores for cover images (e.g. play_q75_cover.csv) is first imported. All columns except for the values are removed from this file; the scores are then moved to the first column. The value 0 is entered in the second column, as these entries are cover images. The corresponding CSV file with the scores of the stego images (e.g. play_q75_stego.csv) is then opened. The column with the scores is copied and appended to the cover entries in the first file. However, the value 1 is entered in the adjacent column for these images, as they have been steganographically manipulated. A header is then inserted above the first line, with the value score written in the first column and the value label in the second column. After saving, a manual check is performed with a text editor to ensure that there are no invisible characters or unwanted formatting that are not displayed by the editing program. Once the files have been checked, the Python script is executed with the prepared CSV file as an argument. After a short runtime, the result is generated and saved in a central file. This process was carried out identically for each data record and each quality level.

Definition of Intrinsic Difficulty

The intrinsic difficulty is based on the previously introduced TER and describes the uncertainty of the model when applied to known data sets under CSM-free conditions. The exact definition of this measure is explained in chapter 4. Specifically, this means that no new values need to be calculated to determine the intrinsic difficulty; instead, the TER values already determined are used directly.

Definition of Inconsistency

As with intrinsic difficulty, no new metrics need to be calculated for inconsistency, as the TER values already determined can be used directly. Inconsistency describes the uncertainty of a model in the presence of cover source mismatch, i.e. when a model trained on data from a specific source is applied to data from other sources. The exact definition can be found in chapter chapter 4. Specifically, the TER values that arise when a model is trained on training data A but then tested on data from sources B, C and D are considered. Since all TER results have been documented centrally, the inconsistency can be derived directly from the existing results document.

6. Results

The following sections present the collected values and results in tables and diagrams. The corresponding interpretation and a detailed classification of the results are then provided in chapter chapter 7.

6.1. Results of prediction score analysis using pre-trained models

The following subchapter presents the results of the prediction score analysis for J-MiPOD and J-UNIWARD using Aletheia's pre-trained models. The average values and total error rates are shown.

6.1.1. J-MiPOD

Average values

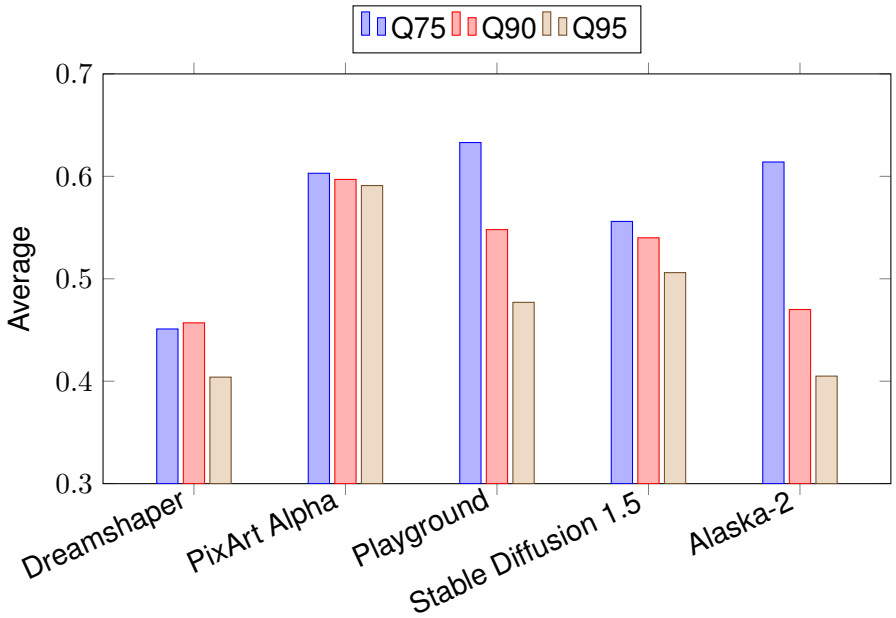
This section contains the results of the analysis of average prediction values when using the pre-trained model based on real data sets from Aletheia. The application was carried out on data sets in which the payload was embedded using J-MiPOD. Table 6.1 shows the average prediction values per data set source across all quality levels. The values are to be understood as the mean values of the respective percentage points.

Table 6.1.: Average Prediction scores at three quality thresholds (10 000 samples each).

Source (10 k)	Q75	Q90	Q95
Dreamshaper	0.451	0.457	0.404
PixArt Alpha	0.603	0.597	0.591
Playground	0.633	0.548	0.477
Stable Diffusion 1.5	0.556	0.540	0.506
Alaska-2	0.614	0.470	0.405

The bar chart 6.1 below shows the values listed in Table 6.1. The data sources are plotted on the x-axis and the respective mean values on the y-axis.

Figure 6.1.: Bar chart presenting average prediction scores at three quality thresholds (10 000 samples each).



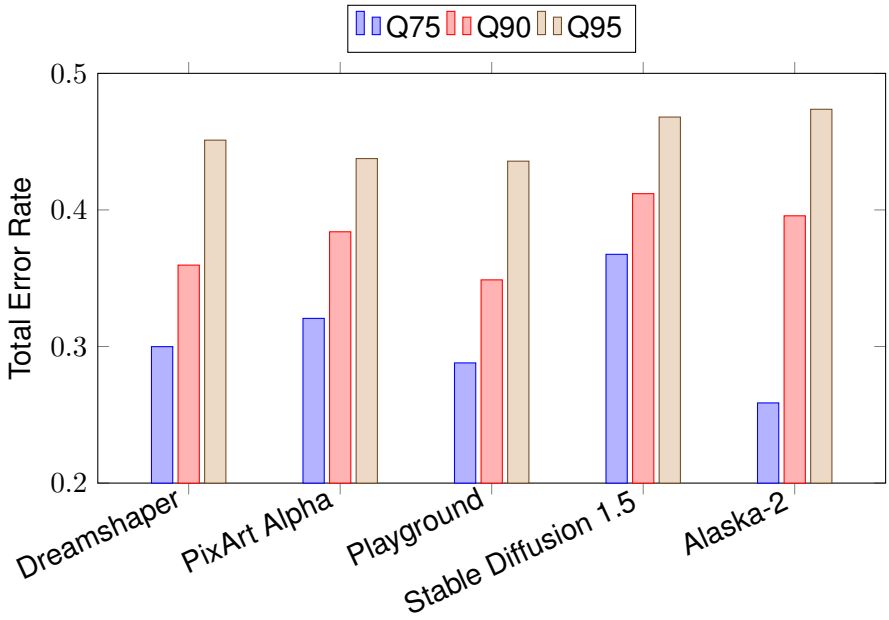
Total Error Rate

The table 6.2 lists the calculated total error rates (TER) determined by applying the Aletheia model to J-MiPOD data sets. The corresponding bar chart visualises these values. All figures are given in percentage points.

Table 6.2.: Total Error Rate of Aletheia J-MiPOD model used on all datasets.

Source (10 k)	Q75	Q90	Q95
Dreamshaper	0.2999	0.3596	0.4511
PixArt Alpha	0.3206	0.384	0.4376
Playground	0.288	0.3488	0.4357
Stable Diffusion 1.5	0.3675	0.4119	0.4680
Alaska-2	0.2587	0.3957	0.4737

Figure 6.2.: Bar chart presenting Total Error Rate of Aletheia J-MiPOD model used on all datasets.



6.1.2. J-UNIWARD

Average values

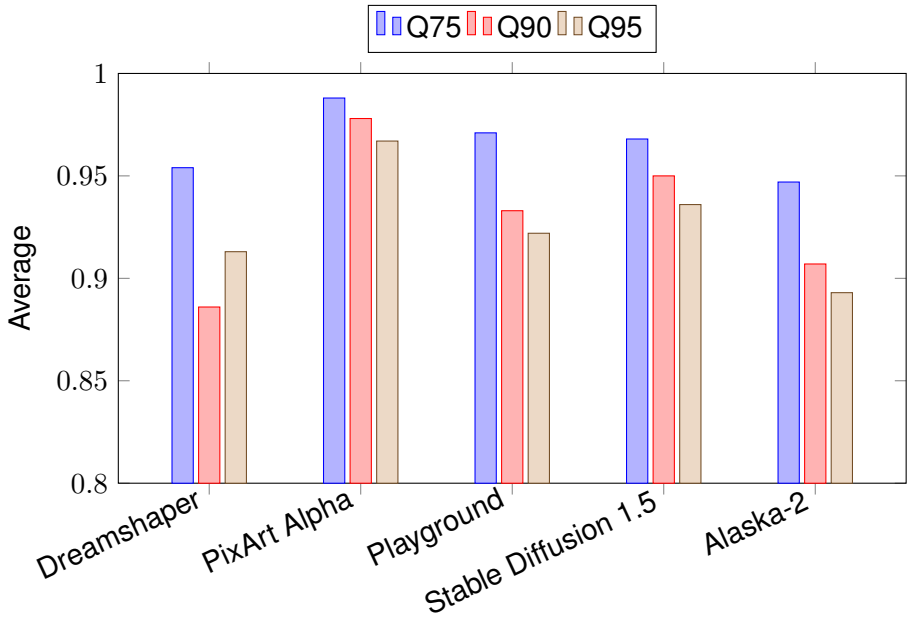
This subsection presents the results obtained from analysing the average values when applying the pre-trained model on real data basis from Aletheia to data sets in which the payload was embedded using J-UNIWARD. Table 6.3 shows the average values for each deepfake data source across all quality levels. The values are to be interpreted as the mean of the respective percentage points.

Table 6.3.: Average Prediction scores at three quality thresholds (10 000 samples each).

Source (10 k)	Q75	Q90	Q95
Dreamshaper	0.954	0.886	0.913
PixArt Alpha	0.988	0.978	0.967
Playground	0.971	0.933	0.922
Stable Diffusion 1.5	0.968	0.950	0.936
Alaska-2	0.947	0.907	0.893

The bar chart 6.3 below shows the results from the table above in a graphical format. The data sources are shown on the x-axis and the results on the y-axis.

Figure 6.3.: Bar chart presenting average prediction scores at three quality thresholds (10 000 samples each).



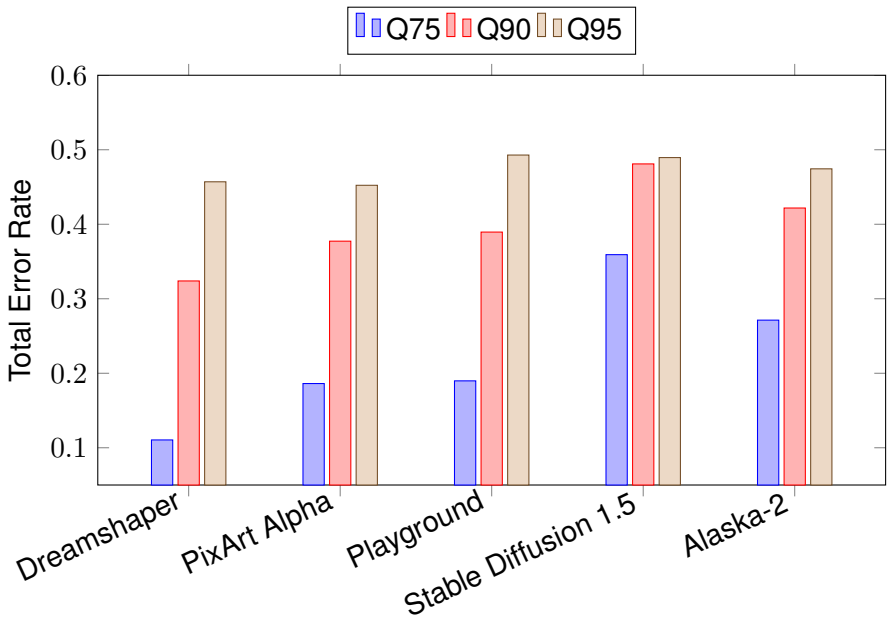
Total Error Rate

Table 6.4 shows the calculated total error rates when applying the pre-trained J-UNIWARD model. The corresponding bar chart 6.4 illustrates these values graphically.

Table 6.4.: Total Error Rate of Aletheia JUNIWARD model used on all datasets.

Source (10 k)	Q75	Q90	Q95
Dreamshaper	0.1105	0.3239	0.4570
PixArt Alpha	0.1862	0.3773	0.4523
Playground	0.1898	0.3895	0.4929
Stable Diffusion 1.5	0.3592	0.4810	0.4895
Alaska-2	0.2713	0.4218	0.4744

Figure 6.4.: Bar chart presenting Total Error Rate of Aletheia J-UNIWARD model used on all datasets.



6.2. Results of the DCI value analysis of the pre-trained model

The following subchapter presents the results of the DCI value analysis for J-MiPOD and J-UNIWARD using Aletheia's pre-trained B models.

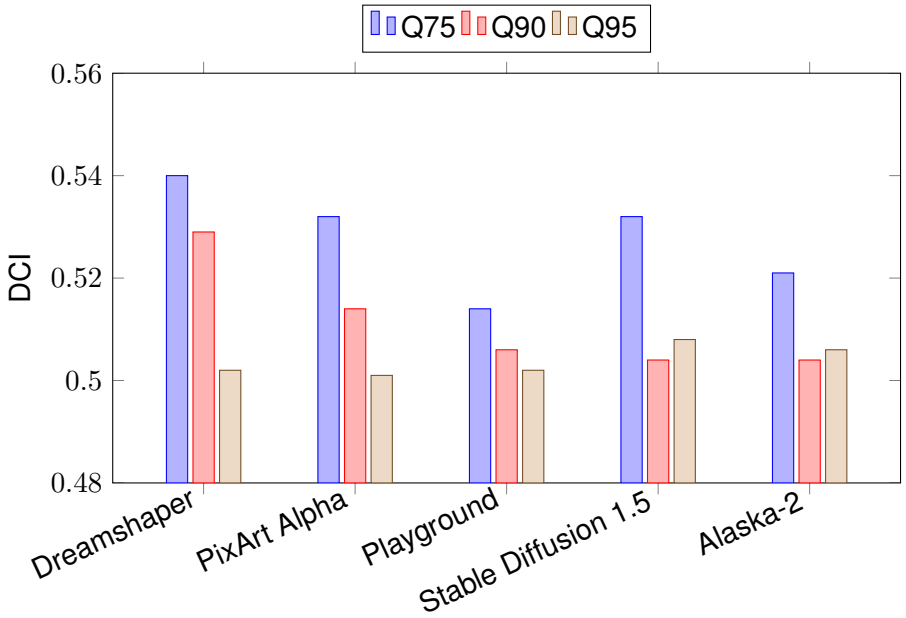
6.2.1. J-MiPOD

Table 6.5 shows the values determined in the DCI (Detection of Classifier Inconsistencies) analysis when using the Aletheia B model on data sets with J-MiPOD embedding. The DCI values indicate the consistency in the analysis of external sources (cover source mismatch). The corresponding results are also visualised in a bar chart 6.5.

Table 6.5.: Detection of Classifier Inconsistencies (DCI) scores at three quality thresholds (400 samples each) $\pm$ 5% error rate

Source	**Q75**	**Q90**	**Q95**
Dreamshaper	0.540	0.529	0.502
PixArt Alpha	0.532	0.514	0.501
Playground	0.514	0.506	0.502
Stable Diffusion 1.5	0.532	0.504	0.508
Alaska-2	0.521	0.504	0.506

Figure 6.5.: Bar chart presenting Detection of Classifier Inconsistencies (DCI) scores at three quality thresholds (400 samples each) $\pm$ 5% error rate.



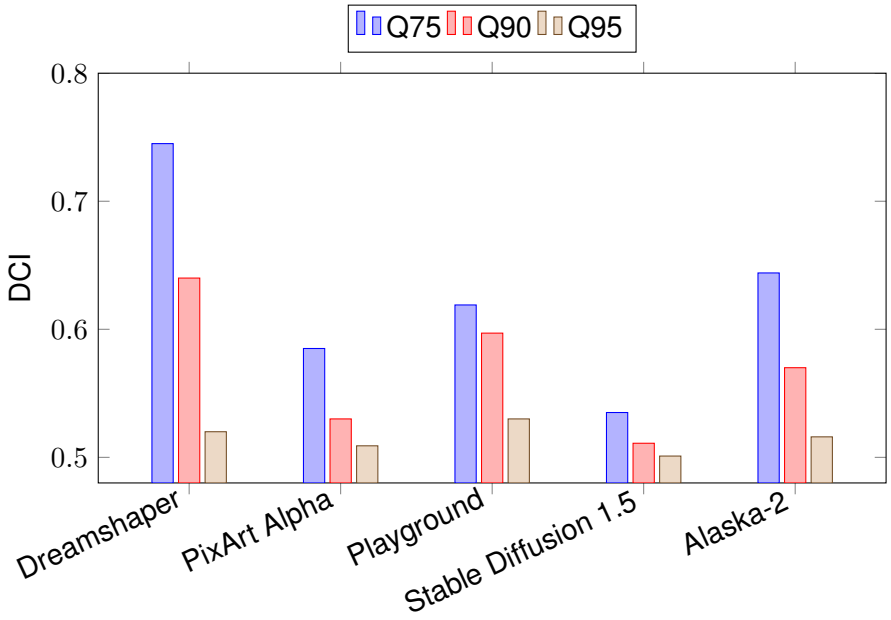
6.2.2. J-UNIWARD

The following table 6.6 and diagram 6.6 show the values of the DCI analysis using Aletheia's B model on J-UNIWARD datasets. These values represent the significance of the Aletheia models used in the analysis of external sources (cover source mismatch).

Table 6.6.: Detection of Classifier Inconsistencies (DCI) scores at three quality thresholds (400 samples each) ± 5% error rate

Source	Q75	Q90	Q95
Dreamshaper	0.745	0.640	0.520
PixArt Alpha	0.585	0.530	0.509
Playground	0.619	0.597	0.530
Stable Diffusion 1.5	0.535	0.511	0.501
Alaska-2	0.644	0.570	0.516

Figure 6.6.: Bar chart presenting Detection of Classifier Inconsistencies (DCI) scores at three quality thresholds (400 samples each) ± 5% error rate.



6.3. Results of prediction score analysis using fine-tuned models based on synthetic sources

The following subchapter presents the results of the prediction score analysis for J-MiPOD and J-UNIWARD using own fine-tuned models. The average values, total error rates, intrinsic difficulty and inconsistency are shown.

6.3.1. J-MiPOD

Average values

The cross-tabulation table 6.7 shows the average prediction values of the analysis using specifically fine-tuned models. These were trained on synthetic deepfake datasets with J-MiPOD embedding. The values refer to each tested combination of model and source as well as to all quality levels (Q75, Q90, Q95). The results should be interpreted as percentage detection probabilities.

Table 6.7.: Average Prediction scores at three quality thresholds (10 000 samples each) for all fine-tuned models on all synthetic sources tested.

	Model											
	Dreamshaper			PixArt Alpha			Playground			Stable Diffusion 1.5		
Source (10k)	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95
Dreamshaper	0.963	0.967	0.940	0.955	0.948	0.930	0.962	0.946	0.901	0.952	0.933	0.873
PixArt Alpha	0.979	0.93	0.736	0.984	0.984	0.956	0.975	0.934	0.85	0.974	0.921	0.766
Playground	0.973	0.938	0.852	0.959	0.865	0.751	0.983	0.976	0.942	0.95	0.91	0.853
Stable Diffusion 1.5	0.965	0.97	0.946	0.948	0.958	0.957	0.951	0.927	0.876	0.963	0.921	0.809

Total Error Rate (TER), Intrinsic Difficulty and Inconsistency

The following cross-tabulation shows the calculated total error rates for J-MiPOD, where each finely tuned model was applied to all sources and all quality levels. In addition, based on the total error rates, the values for intrinsic difficulty (without CSM) are shown in blue and the values for inconsistency (with CSM) are shown in orange. All values are expressed as percentage points.

Table 6.8.: Total Error Rate, Intrinsic Difficulty (marked blue) and Inconsistency (marked orange) values of all fine-tuned models for J-MiPOD used on all deepfake datasets

	Model											
	Dreamshaper			PixArt Alpha			Playground			Stable Diffusion 1.5		
Source (10k)	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95
Dreamshaper	0.0237	0.0229	0.0418	0.0326	0.0517	0.1075	0.0401	0.0555	0.1172	0.0377	0.043	0.0653
PixArt Alpha	0.0407	0.0645	0.1485	0.0055	0.0117	0.0472	0.0464	0.1197	0.2326	0.0298	0.0531	0.1094
Playground	0.029	0.0505	0.1079	0.0323	0.0954	0.1974	0.0154	0.0215	0.0485	0.0279	0.0438	0.0665
Stable Diffusion 1.5	0.0882	0.1711	0.2961	0.0807	0.2008	0.3345	0.0847	0.1642	0.3114	0.0331	0.0523	0.1232

6.3.2. J-UNIWARD

Average values

The cross-tabulation table 6.7 shows the average prediction values of the analysis using specifically fine-tuned models. These were trained on synthetic deepfake datasets with J-UNIWARD embedding. The values refer to each tested combination of model and source as well as to all quality levels (Q75, Q90, Q95). The results should be interpreted as percentage detection probabilities.

Table 6.9.: Average Prediction scores at three quality thresholds (10 000 samples each) for all fine tuned models on all synthetic sources tested.

	Model											
	Dreamshaper			PixArt Alpha			Playground			Stable Diffusion 1.5		
Source (10k)	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95
Dreamshaper	0.983	0.954	0.9	0.984	0.952	0.897	0.909	0.897	0.798	0.898	0.847	0.768
PixArt Alpha	0.969	0.868	0.747	0.989	0.973	0.915	0.921	0.839	0.739	0.948	0.848	0.689
Playground	0.962	0.865	0.799	0.969	0.925	0.87	0.964	0.965	0.935	0.874	0.777	0.721
Stable Diffusion 1.5	0.987	0.961	0.937	0.996	0.977	0.952	0.928	0.916	0.854	0.927	0.877	0.764

Total Error Rate (TER), Intrinsic Difficulty and Inconsistency

The following cross-tabulation shows the calculated total error rates for J-UNIWARD, where each finely tuned model was applied to all sources and all quality levels. In addition, based on the total error rates, the values for intrinsic difficulty (without CSM) are shown in blue and the values for inconsistency (with CSM) are shown in orange. All values are expressed as percentage points.

Table 6.10.: Total Error Rate, Intrinsic Difficulty (marked blue) and Inconsistency (marked orange) values of all fine-tuned models for J-UNIWARD used on all deepfake datasets

	Model											
	Dreamshaper			PixArt Alpha			Playground			Stable Diffusion 1.5		
Source (10k)	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95	Q75	Q90	Q95
Dreamshaper	0.0231	0.0475	0.084	0.0464	0.092	0.1415	0.0405	0.0956	0.1708	0.0633	0.0805	0.1126
PixArt Alpha	0.039	0.0787	0.1379	0.0095	0.0239	0.0632	0.0325	0.0912	0.1641	0.0448	0.0755	0.1287
Playground	0.0607	0.1197	0.1709	0.0622	0.1198	0.169	0.0158	0.0354	0.0743	0.0815	0.1199	0.1623
Stable Diffusion 1.5	0.1008	0.2311	0.2834	0.153	0.2387	0.2868	0.1316	0.234	0.2878	0.0663	0.1167	0.1756

7. Discussion

This chapter first interprets and discusses the results and answers the research questions. It then presents experiences and opinions on the success of the work.

7.1. Robustness of pre-trained models against cover source mismatch

The first research question addresses the extent to which a steganography analysis model trained on real JPEG images is robust against cover source mismatch (CSM) when applied to synthetically generated, coloured JPEG images with different compression levels. For this analysis, pre-trained models from the Aletheia platform were used, which were trained on the extensive, heterogeneous real image dataset of the Alaska-2 Challenge. The embedding rate used there averaged 0.4 bpnzAC in the original training, but with variable distribution across the individual images. Similarly, the JPEG images were processed in the training with randomly selected compression levels (Q75, Q90 or Q95). In the present work, however, all images used - both synthetic and real - were systematically exported in all three quality levels and the embedding rate was set to a constant 0.4 bpnzAC. This results in a homogeneous distribution of noise information, which differs significantly from the training setup. The results achieved show that the pre-trained models are only robust to cover source mismatch to a limited extent. The average prediction probabilities varied significantly depending on the image source, the JPEG quality level and the embedding method used (J-MiPOD vs. J-UNIWARD). For J-MiPOD, the values for synthetic sources such as Dreamshaper or Playground, as well as for Alaska-2, the nominally known dataset, were consistently at or below 0.63 - with a minimum of 0.404 for Dreamshaper (Q95) and 0.405 for Alaska-2 (Q95). The observation that the model Alaska-2 with a fixed embedding rate was unable to classify known sources is particularly revealing: despite identical image content but different embedding and compression distributions, the model showed no discernible performance advantages over unknown sources.

This suggests that differences in noise distribution - with constant semantics - can also act as a form of CSM. The model performance therefore depends not only on the image source in the narrow sense, but also on statistical differences in the structure of the input data. A clear influence of JPEG compression is also apparent: With increasing quality factor (Q75 -> Q95), the detection probability decreased almost continuously. Since the embedding rate was constant across all experiments, these differences are not due to the strength of the embedding, but primarily to the reduced interference effectiveness at higher image quality and the interaction with the source structure. The Total Error Rate (TER) confirms these observations: While values below 0.35 were still measured in many cases at Q75, the TER values regularly rose above 0.45 at Q95. Alaska-2 also showed a TER of 0.4737 at Q95 - almost identical to Stable Diffusion 1.5 (0.4680). This once again illustrates that supposed familiarity with a dataset does not guarantee robust recognition performance if hidden data characteristics (such as compression or embedding distribution) have changed. For further classification, the metric Detection of Classifier Inconsistencies (DCI) was used, which measures the stability of model decisions when sources change. For J-MiPOD, the DCI values remained constant in the range of 0.50 to 0.54, almost independently of quality. These values indicate uncertain and inconsistent model decisions as soon as new sources are processed, even those with minimal formal changes. The low DCI even for Alaska-2 proves that the models do not generalise stably, even when the image content is known. In contrast, application to J-UNIWARD datasets showed significantly higher prediction probabilities, with values between 0.93 and 0.99 at Q75. However, here too, performance declined with increasing quality, albeit at a higher overall level. The TER values were below 0.20 in many cases at Q75 and also rose to above 0.45 at Q95. The DCI values showed a more differentiated picture: at Q75, they were sometimes above 0.70 (e.g. Dreamshaper: 0.745), which suggests more consistent classifications at low compression. At Q95, however, the DCI values also dropped to a level close to that of J-MiPOD. In summary, it can be concluded that the pre-trained models examined - despite their training on a large, realistic dataset - are not sufficiently robust against cover source mismatch when applied to synthetically generated image sources or modified variants of known data. Both the choice of embedding method and the compression level and distribution of the interference information significantly influence model performance. Especially with high JPEG quality (Q95) and simultaneously deviating source statistics, high error rates and inconsistent decisions occur. These findings highlight the need to consider not only the source semantics but also the distribution statistics of the input data when evaluating steganography analysis models. Transfer-

ring pre-trained models to new image sources is therefore only recommended if their statistical structure was sufficiently represented in the training.

7.2. Influence of fine-tuning on synthetically generated data sets

The second research question investigated the extent to which targeted fine-tuning of models on synthetically generated datasets can improve classification performance, particularly with regard to the problem of cover source mismatch (CSM). To this end, one model was trained for each synthetic source and embedding method (J-MiPOD, J-UNIWARD). The training data comprised all JPEG quality levels (Q75, Q90, Q95) in mixed form, while the embedding rate was set to a constant 0.4 bpnzAC. Analysis of the average prediction values shows that fine-tuned models are capable of achieving significantly higher detection probabilities than the pre-trained base models. When applied to the test data matching the source, the detection values were consistently above 0.95, in some cases even above 0.98 (e.g. PixArt model on PixArt Q75: 0.984). At the same time, it can be seen that the model performance strongly depends on the match between the training and test sources: When the source differed and the JPEG quality was high, the detection values sometimes fell below 0.80. This demonstrates the high source specificity of the models - a property that can lead to limited transferability in the absence of domain coverage. The total error rate (TER) also reflects this correlation: For test data from the same source as the training model, the TER values were often below 0.05 - an indication of consistent decision-making behaviour with stable data distribution. Under CSM conditions, however, and especially at Q95, the error rates increased significantly, in some cases to over 0.25 (e.g. playground model on Stable Diffusion at Q95). In order to differentiate between inherent classification difficulty and CSM-related deterioration, the intrinsic difficulty (ID) and inconsistency (IC) were also calculated. The ID values remained low (< 0.05) for suitable model-source combinations, indicating a fundamentally solvable task given the structure. The IC values, on the other hand - i.e. the increase in errors caused by cover source mismatch - showed strong differences depending on the model and source. In some cases, the IC was above 0.20, indicating a pronounced sensitivity to domain-alien structures. It is also noteworthy that these correlations cannot be explained by differences in the embedding rate, as this was kept the same in all constellations. The observed performance differences can therefore be attributed primarily to the statistics of the image source, the model architecture and the complexity of the embedding (depending on the JPEG compression).

Overall, it can be concluded that:

- fine-tuning on synthetic sources can substantially increase detection performance,
- however, the models are highly adapted to the source distribution,
- and under CSM conditions - especially with high JPEG quality - significant performance drops can occur.

The results show that higher prediction probabilities have a positive effect on classification confidence, but must always be considered in the context of application-specific requirements and defined thresholds. A high detection probability is not required in every scenario, but depends on the tolerance for false positive and false negative rates. For robust, generalisable models, a more cross-domain training approach will be necessary in the future - for example, through the aggregation of synthetic sources or domain adaptation methods.

7.3. Comparison with Méreur et al. (2024) and classification of own results

The study [1] by Méreur et al. (2024) is one of the first systematic investigations into the use of synthetically generated deepfakes as carrier material for steganography and analyses their impact on the effectiveness of modern steganography analysis methods, taking into account the Cover Source Mismatch (CSM) [1]. The present work also focuses on the problem of CSM, although despite the common objective, there are several significant differences in methodology, depth of analysis and interpretation of results. A first fundamental difference lies in the choice and treatment of image sources: Méreur et al. conduct a comparative evaluation of several generic sources - including real images and synthetically generated deepfakes - but largely consider deepfake generators as a homogeneous class. In contrast, our own work takes a fine-grained, source-specific approach: dedicated training and test sets were created and analysed in detail for four different synthetic image sources (Dreamshaper, PixArt Alpha, Playground, Stable Diffusion 1.5). This atomistic approach not only allows the evaluation of the general CSM effect, but also the identification of particularly problematic generators in terms of their detectability by steganalytic models (6.3). A second key difference concerns image compression: while Méreur et al. primarily consider uncompressed or unspecified image formats, the focus of this work is explicitly on the JPEG compression process. [1] The investigation of different quality levels (Q75, Q90, Q95) shows that higher compression (e.g. Q75) leads to significantly improved detection performance, while recognition

rates decrease with higher image quality (6.1,6.3) - an aspect that is not addressed in [1]. There are also significant differences in terms of model strategy: Méreur et al. analyse only generically pre-trained models and show that their detection performance decreases significantly with CSM - with total error rates (TER) of up to 50%, which indicates a virtually random classification. [1] This observation is consistent with the results of our own work, in which Aletheia models pre-trained on the ALASKA-2 dataset show similarly high error rates on synthetic test data (6.1). In addition, however, targeted fine-tuning was performed on artificial training data. The results show that fine-tuned models achieve significantly better performance - especially in terms of reducing TER and consistency across different synthetic sources. For example, when using J-MiPOD on PixArt Alpha images, the TER was reduced from over 45% (pre-trained) to less than 20% (fine-tuned). Both works use the metrics Intrinsic Difficulty and Inconsistency for the quantitative evaluation of the CSM problem. While Méreur et al. aggregate the values and use them to visualise the effect between sources, the present work differentiates these metrics by source and quality level. [1] This shows that the intrinsic difficulty varies significantly depending on the generator, which indicates different statistical properties of the synthetic sources. The inconsistency between sources is also considerable in some cases, which underlines the challenge of generalisation. In summary, the two studies complement each other in terms of content: while [1] examine the CSM effect from a fundamental, broad perspective and highlight the fundamental vulnerability to synthetic image sources, our own work provides a practical, methodologically in-depth analysis with concrete optimisation strategies. Both studies highlight the relevance of synthetic images as a serious challenge for existing steganography analysis methods. The results presented here expand on this understanding by showing that targeted fine-tuning on synthetic sources can significantly improve detection performance and increase generalisability across generator boundaries.

7.4. Overall experience and thoughts to the reserach

It was particularly positive that access to suitable models, tools and information was much easier than expected. Platforms such as Hugging Face enabled quick and easy access to powerful models, while the generation of synthetic image datasets proved to be more intuitive and resource-efficient than anticipated. Although the field is still very young, especially in connection with AI-generated images, there is already a broad body of scientific work that has greatly facilitated the acquisition of information. At the same time, unexpected challenges arose in the course

of practical implementation. During data generation, NSFW content was occasionally generated unintentionally, requiring subsequent cleanup, which delayed progress. In addition, technical problems - in particular driver incompatibilities during GPU updates - repeatedly led to interruptions. In retrospect, some decisions to continue experiment series were made too early without sufficiently critically assessing the potential relevance of the results; this made it necessary to perform some calculations multiple times. Despite these operational difficulties, the overall objective of the master's thesis was fully achieved. More time and additional computing resources would have allowed for further exploration of other aspects of the research (see Limitations), but the present work already provides a well-founded overview and valuable insights that can be directly incorporated into further research. With the chosen scope - the combination of synthetic deepfakes, coloured JPEG domain, steganalysis using models and an atomistic fine-tuning approach - clear contributions to the current state of research have been achieved. While existing work focuses predominantly on the spatial domain, this thesis specifically covers the JPEG domain and thus expands existing knowledge. The results achieved provide a robust foundation on which future research projects can build directly.

8. Conclusion

The aim of this work was to investigate the effectiveness and robustness of steganography analysis models based on real data in detecting steganographic content in artificially generated deepfake images, with a particular focus on the coloured JPEG compression domain and possible effects of cover source mismatch (CSM). To this end, Aletheia [28] models pre-trained on real images were applied to synthetic deepfake datasets at various compression levels (Q75, Q90, Q95), and domain-specific models were trained specifically on synthetic data sources using atomistic fine-tuning. The results show that although the pre-trained models have solid basic recognition capabilities, they lose significant performance when applied to image sources whose statistical structure differs from the training data due to embedding parameters, generator architecture or compression distribution. This effect also occurs when the source base is formally the same (e.g., Alaska-2), illustrating that CSM arises not only from different image content, but also from seemingly subtle variations in the image generation chain. Particularly high JPEG qualities (Q95) and embedding methods with weak noise signals (e.g. J-MiPOD) lead to significant performance drops and inconsistent model decisions. Targeted fine-tuning to synthetic sources significantly improved detection performance, especially when training and test data came from the same source. At the same time, however, the models showed strong source specificity: even slight changes in the image source or compression level led to increased error rates. Even fine-tuning did not allow true cross-domain generalisability to be fully achieved. Overall, this work confirms that classic steganography analysis methods are fundamentally capable of identifying steganographic content in coloured JPEG deepfakes - but only if the statistical structure of the analysed data was sufficiently represented in the training. This means that careful design of the training data and evaluation of domain-specific influencing factors (e.g. compression, generator architecture, embedding method) are of central importance. For future research, there are numerous opportunities to deepen these findings, provided that the time and computing resources are expanded accordingly: A systematic comparative analysis of alternative steganography analysis methods, investi-

gations of other image resolutions, greyscale artificial images, extreme compression levels, and other deepfake generators and embedding methods (e.g. GANs or nsF5) could make significant contributions to understanding the interaction between data source, embedding, and detection.

8.1. Future Work

Several limitations arose in the course of this work, the influence of which on the results achieved can only be estimated to a limited extent without further investigation. However, these limitations also offer a wide range of starting points for future research projects. One key aspect concerns the choice of steganography analysis tool. This study used only one specific, publicly available, user-friendly steganography analysis tool with pre-trained models. Although this approach enables reproducible and technically accessible analysis, the results are therefore limited to this one tool. Further research should therefore conduct a comparative evaluation with other established steganography analysis methods in order to verify the generalisability of the results and assess their robustness against different detection approaches. Another limiting factor was the restriction to images with a fixed resolution of 512x512 pixels. This specification was made for practical reasons and to standardise the test environment. However, it is known that larger image dimensions offer a higher capacity for embedding useful data, which in turn can influence detectability. In addition, the analysis model used requires large-format images to be divided into sub-segments, the individual results of which are then aggregated. This can lead to altered detection characteristics, especially with regard to fine-grained embedding patterns. Future studies could therefore investigate the extent to which image size affects the detection performance and effectiveness of different steganography analysis models. Another aspect that has not yet been explored in depth concerns the behaviour of steganography analysis on artificially generated grayscale images. This work focused on colour images that were additionally processed using lossy compression - a scenario that is considered particularly challenging for steganography analysis. Nevertheless, grayscale images are a relevant field of application for steganographic methods, and there are generator models specifically designed for grayscale images. Future work could specifically investigate whether and to what extent steganography analysis differs on grayscale images, both in terms of detection accuracy and in terms of the requirements for training and detection models. Another promising area of investigation concerns images with particularly high compression, for example due to extremely low JPEG quality settings. In this work, moderate compression was used, but a greater reduction

in image quality could significantly alter the characteristics of embedded payloads and thus also make detection more difficult or easier. A systematic analysis of such scenarios would therefore be of great value. Finally, it should be noted that this work was limited exclusively to images generated by diffusion models. Other relevant image generation approaches, such as Generative Adversarial Networks (GANs), were deliberately excluded. Similarly, the use of alternative steganographic embedding methods - such as nsF5, UERD or StegHide - was not considered. Extending the analysis to these techniques could provide valuable insights into the interaction between generative image models, embedding methods and detection techniques, thus making a further important contribution to the advancement of robust steganography analysis.

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
bpnzAC	bites per non-zero AC DCT coefficient
CNN	Convolutional neural network
CPU	Central processing unit
CSV	Comma-separated values
dB	Decibel
DCI	Detection of Classifier Inconsistencies
DCT	Discrete cosine transform
GAN	Generative Adversarial Network
GB	Gigabyte
GPU	Graphics processing unit
JMIPOD or J-MiPOD	Minimizing Performance of Optimal Detector in JPEG domain
JPEG	Joint Photographic Experts Group
JUNIWARD or J-UNIWARD	Universal wavelet relative distortion in JPEG domain
KB	Kilobyte

LSB	least significant bit
MCU	Minimum Coded Unit
NSFW	Not safe for work
PNG	Portable network graphics
PSNR	Peak signal-to-noise ratio
RAM	Random access memory
TIFF or TIF	Tagged Image File Format
VRAM	Video random-access memory

Bibliography

[1] Arthur Méreur, Antoine Mallet, and Rémi Cogranne, “Are deepfakes a game-changer in digital images steganography leveraging the cover-source-mismatch?” In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ser. ARES ’24, Vienna, Austria: Association for Computing Machinery, 2024, ISBN: 9798400717185. DOI: 10.1145/3664476.3670893.

[2] R.J. Anderson and F.A.P. Petitcolas, “On the limits of steganography,” *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 4, pp. 474–481, 1998. DOI: 10.1109/49.668971.

[3] Catherine Taylor Clelland, Viviana Risca, and Carter Bancroft, “Hiding messages in dna microdots,” *Nature*, vol. 399, no. 6736, pp. 533–534, Jun. 1999. DOI: 10.1038/21092.

[4] Fabien Petitcolas, Ross Anderson, and Markus Kuhn, “Information hiding - a survey,” *Proceedings of the IEEE*, vol. 87, pp. 1062–1078, Aug. 1999. DOI: 10.1109/5.771065.

[5] Kushalveer Singh Bachchas. “Lsb steganography: Hiding confidential data within pictures.” abgerufen am 6. Juli 2025.

[6] Syed Rifat Raiyan and Md. Hasanul Kabir, *Screedsolo: A secure and robust lsb image steganography framework*, arXiv:2503.12368, 2025.

[7] Y. Y. Ghadi, T. AlShloul, Z. I. Nezami, H. Ali, M. Asif, and M. Jaward Bah, “Enhanced payload volume in the least significant bits image steganography using hash function,” *PeerJ Computer Science*, vol. 9, e1606, 2023. DOI: 10.7717/peerj-cs.1606.

[8] A. Nagalinga Rajan and P. Eswaran, “High capacity robust image steganography in the dct domain using spread spectrum technique,” *International Journal of Applied Engineering Research*, vol. 10, no. 6, pp. 14 489–14 496, 2015, ISSN: 0973-4562.

[9] Stuti Goel, Arun Kumar Rana, and Manpreet Kaur, “A dct-based robust methodology for image steganography,” *www.ijcst.com*, vol. 4, Jul. 2013.

[10] Zaid Al-Omari and Ahmad Al-Taani, “A survey on digital image steganography,” May 2015. DOI: 10.15849/icit.2015.0016.

[11] Shahid Rahman, Jamal Uddin, Hameed Hussain, Sabir Shah, Abdu Salam, Farhan Amin, Isabel de la Torre Díez, Debora Libertad Ramírez Vargas, and Julio César Martínez Espinosa, “A novel and efficient digital image steganography technique using least significant bit substitution,” *Scientific Reports*, vol. 15, no. 1, p. 107, Jan. 2025, ISSN: 2045-2322. DOI: 10.1038/s41598-024-83147-3.

[12] Jessica Fridrich, Miroslav Goljan, and Dorin Hogea, “Steganalysis of jpeg images: Breaking the f5 algorithm,” in *Information Hiding*, Fabien A. P. Petitcolas, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 310–323, ISBN: 978-3-540-36415-3.

[13] Jimin Zhang, Xianfeng Zhao, and Xiaolei He, “Robust jpeg steganography based on the robustness classifier,” *EURASIP Journal on Information Security*, vol. 2023, no. 1, p. 11, 2023. DOI: 10.1186/s13635-023-00148-x.

[14] Amitava Nag, Sushanta Biswas, Debasree Sarkar, and Partha Pratim Sarkar, “A novel technique for image steganography based on block-dct and huffman encoding,” *CoRR*, vol. abs/1006.1186, 2010. arXiv: 1006.1186. [Online]. Available: http://arxiv.org/abs/1006.1186.

[15] G.K. Wallace, “The jpeg still picture compression standard,” *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992. DOI: 10.1109/30.125072.

[16] Rémi Cogranne, Eva Giboulot, and Patrick Bas, “Efficient steganography in jpeg images by minimizing performance of optimal detector,” *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 1328–1343, 2022. DOI: 10.1109/TIFS.2021.3111713.

[17] Vojtech Holub, Jessica Fridrich, and Tomas Denemark, “Universal distortion function for steganography in an arbitrary domain,” *EURASIP Journal on Information Security*, vol. 2014, no. 1, p. 1, Jan. 2014, ISSN: 1687-417X. DOI: 10.1186/1687-417X-2014-1.

[18] Antoine Mallet, Martin Bene, and Rémi Cogranne, “Cover-source mismatch in steganalysis: Systematic review,” *EURASIP Journal on Information Security*, vol. 2024, 2024. DOI: 10.1186/s13635-024-00171-6.

[19] Quentin Giboulot, Patrick Bas, Rémi Cogranne, and Dirk Borghys, "The cover source mismatch problem in deep-learning steganalysis," in *Proceedings of the 30th European Signal Processing Conference (EUSIPCO 2022)*, Belgrade, Serbia: IEEE, 2022, pp. 1032–1036, ISBN: 978-1-6654-6798-8. DOI: 10.23919/EUSIPCO55093.2022.9909553.

[20] Dominik Sepak, Lukas Adam, and Tomas Pevny, "Formalizing coversource mismatch as a robust optimization," in *Proceedings of the 30th European Signal Processing Conference (EUSIPCO 2022)*, Belgrade, Serbia: IEEE, Aug. 2022, pp. 1042–1046, ISBN: 978-1-6654-6798-8.

[21] Dirk Borghys, Patrick Bas, and Helena Bruyninckx, "Facing the cover-source mismatch on jphide using training-set design," in *Proceedings of the 6th ACM Workshop on Information Hiding and Multimedia Security*, ser. IHMMSec '18, Innsbruck, Austria: Association for Computing Machinery, 2018, pp. 17–22, ISBN: 9781450356251. DOI: 10.1145/3206004.3206021.

[22] Quentin Giboulot, Rémi Cogranne, Dirk Borghys, and Patrick Bas, "Effects and solutions of cover-source mismatch in image steganalysis," *Signal Processing: Image Communication*, vol. 86, p. 115 888, 2020, ISSN: 0923-5965. DOI: 10.1016/j.image.2020.115888. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0923596520300941.

[23] Dominik epák, Luká Adam, and Tomá Pevn, "Formalizing cover-source mismatch as a robust optimization," in *EUSIPCO: European Signal Processing Conference, Belgrade, Serbia*, 2022.

[24] Giboulot Quentin, Bas Patrick, Cogranne Rémi, and Borghys Dirk, "The cover source mismatch problem in deep-learning steganalysis," in *2022 30th European Signal Processing Conference (EUSIPCO)*, 2022, pp. 1032–1036. DOI: 10.23919/EUSIPCO55093.2022.9909553.

[25] Antoine Mallet, Rémi Cogranne, Minoru Kuribayashi, Arthur Méreur, et al., "How much is the source mismatch an important problem for deepfake detection?" *APSIPA Transactions on Signal and Information Processing*, vol. 14, no. 3, 2025.

[26] Mehdi Boroumand and Jessica Fridrich, "Scalable processing history detector for jpeg images," *Electronic Imaging*, vol. 29, no. 7, pp. 128–128, 2017. DOI: 10.2352/ISSN.2470-

1173.2017.7.MWSF-336. [Online]. Available: https://library.imaging.org/ei/articles/29/7/art00019.

[27] Martin Benes, Nora Hofer, and Rainer Böhme, "The effect of the jpeg implementation on the cover-source mismatch error in image steganalysis," in *2022 30th European Signal Processing Conference (EUSIPCO)*, 2022, pp. 1057–1061. DOI: 10.23919/EUSIPCO55093.2022.9909711.

[28] Daniel Lerch-Hostalot and David Megías, "Aletheia: an open-source toolbox for steganalysis," *Journal of Open Source Software*, vol. 9, no. 93, p. 5982, Jan. 2024. DOI: 10.21105/joss.05982. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.05982.

[29] Remi Cogranne, Quentin Giboulot, and Patrick Bas, "Alaska2 dataset," Dataset, Creative Commons BY–NC–ND licence. Zugriff am 13. Juli 2025, 2020. Accessed: Jul. 13, 2025.

[30] Cole Stryker. "What is hugging face?" Abgerufen am 3. Juli 2025. [Online]. Available: https://www.ibm.com/think/topics/hugging-face.

[31] Lykon, *Dreamshaper 8*, https://huggingface.co/Lykon/dreamshaper-8, Zugriff am 03. 07. 2025, 2023.

[32] Lykon, *Dreamshaper v8.0 - model card*, https://www.comflowy.com/model/dreamshaper-v-8, Zugriff am 03. 07. 2025, 2023.

[33] kidelaleron, *Dreamshaper v8 - maybe the end of the journey*, https://www.reddit.com/r/StableDiffusion/comments/15d2uc2, Zugriff am 03. 07. 2025, 2023.

[34] PixArt-alpha, *Pixart-xl-2-512x512*, https://huggingface.co/PixArt-alpha/PixArt-XL-2-512x512, Zugriff am 03. 07. 2025, 2023.

[35] Junsong Chen, Jincheng Yu, Chongjian Ge, Lewei Yao, Enze Xie, Yue Wu, Zhongdao Wang, James Kwok, Ping Luo, Huchuan Lu, and Zhenguo Li, *Pixart-α: Fast training of diffusion transformer for photorealistic text-to-image synthesis*, 2023. arXiv: 2310.00426 [cs.CV].

[36] Junsong Chen, Yue Wu, Simian Luo, Enze Xie, Sayak Paul, Ping Luo, Hang Zhao, and Zhenguo Li, *Pixart-δ: Fast and controllable image generation with latent consistency models*, 2024. arXiv: 2401.05252 [cs.CV].

[37] Daiqing Li, Aleks Kamko, Ali Sabet, Ehsan Akhgari, Lin Xu, and Suhail Doshi, *Playground v2*. [Online]. Available: https://huggingface.co/playgroundai/playground-v2-1024px-aesthetic.

[38] The Decoder. "Playground v2 is a new text-to-image model that competes with stable diffusion xl." Zugriff am 03.07.2025. [Online]. Available: https://the-decoder.com/playground-v2-is-a-new-text-to-image-model-that-competes-with-stable-diffusion-xl.

[39] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Bjoern Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2022, pp. 10 684–10 695.

[40] Vojtch Holub and Jessica Fridrich, "Phase-aware projection model for steganalysis of jpeg images," in *Media Watermarking, Security, and Forensics 2015*, Adnan M. Alattar, Nasir D. Memon, and Chad D. Heitzenrater, Eds., International Society for Optics and Photonics, vol. 9409, SPIE, 2015, 94090T. DOI: 10.1117/12.2075239.

[41] Shunquan Tan, Qiushi Li, Laiyuan Li, Bin Li, and Jiwu Huang, "Std-net: Search of image steganalytic deep-learning architecture via hierarchical tensor decomposition," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2657–2673, 2023. DOI: 10.1109/TDSC.2023.3267065.

[42] Daniel Lerch-Hostalot. "About daniel lerch hostalot." Zugriff am 3. Juli 2025. [Online]. Available: https://daniellerch.me/about-en/.

[43] David Megías Jiménez. "Expertenprofil an der uoc." Zugriff am 3. Juli 2025. [Online]. Available: https://www.uoc.edu/en/news/media-services/experts-guide/david-megias.

[44] Rémi Cogranne, Quentin Giboulot, and Patrick Bas, "Steganography by minimizing statistical detectability: The cases of jpeg and color images," in *Proceedings of the 2020 ACM Workshop on Information Hiding and Multimedia Security*, ser. IHMMSec '20, Denver, CO, USA: Association for Computing Machinery, 2020, pp. 161–167, ISBN: 9781450370509. DOI: 10.1145/3369412.3395075.

[45] Vahid Sedighi, Rémi Cogranne, and Jessica Fridrich, “Content-adaptive steganography by minimizing statistical detectability,” *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 221–234, 2016. DOI: 10.1109/TIFS.2015.2486744.

[46] Mingxing Tan and Quoc Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in *Proceedings of the 36th International Conference on Machine Learning*, Kamalika Chaudhuri and Ruslan Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Jun. 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html.

A. Appendix Code and Scripts

This appendix offers all code and scripts used in this work.

A.1. Code for Generators

A.1.1. dreamshaper_batch.py

```python

from diffusers import DiffusionPipeline

# Load the model
model_id = "Lykon/dreamshaper-8"
pipeline = DiffusionPipeline.from_pretrained(model_id)

# Set up the prompt and device
prompt = ""
pipeline = pipeline.to("cuda")

# Loop to generate 10 000 images
for i in range(1, 10_001):
    # Generate the image
    image = pipeline(prompt).images[0]

    # Save the image with a count-based filename
    image.save(f"image_{i}.png")

    print(f"Saved image_{i}.png")
```

Listing A.1: Python script for generating 10,000 images with diffusers and Dreamshaper model

1

A.1.2. pixart_vram_optimized.py

```
# pip install -U accelerate transformers bitsandbytes
# pip install -U git+https://github.com/huggingface/diffusers

from transformers import T5EncoderModel
from diffusers import PixArtAlphaPipeline
import torch
import gc


def flush():
    gc.collect()
    torch.cuda.empty_cache()

def bytes_to_giga_bytes(bytes):
    return bytes / 1024 / 1024 / 1024

# Loading in 8 bits needs `bitsandbytes`.
text_encoder = T5EncoderModel.from_pretrained(
    "PixArt-alpha/PixArt-XL-2-1024-MS",
    subfolder="text_encoder",
    load_in_8bit=True,
    device_map="auto",
)

pipe = PixArtAlphaPipeline.from_pretrained(
    "PixArt-alpha/PixArt-XL-2-1024-MS",
    text_encoder=text_encoder,
    transformer=None,
    device_map="auto"
)

```

[1] Created with help from ChatGPT o3 and ChatGPT 4o

```
with torch.no_grad():
    prompt = "cute cat"
    prompt_embeds, prompt_attention_mask, negative_embeds,
        negative_prompt_attention_mask = pipe.encode_prompt(prompt)

del text_encoder
del pipe
flush()

pipe = PixArtAlphaPipeline.from_pretrained(
    "PixArt-alpha/PixArt-XL-2-1024-MS",
    text_encoder=None,
    torch_dtype=torch.float16,
).to("cuda")

latents = pipe(
    negative_prompt=None,
    prompt_embeds=prompt_embeds,
    negative_prompt_embeds=negative_embeds,
    prompt_attention_mask=prompt_attention_mask,
    negative_prompt_attention_mask=negative_prompt_attention_mask,
    num_images_per_prompt=1,
    output_type="latent",
).images

del pipe.transformer
flush()

with torch.no_grad():
    image = pipe.vae.decode(latents / pipe.vae.config.scaling_factor,
        return_dict=False)[0]
image = pipe.image_processor.postprocess(image, output_type="pil")

image[0].save("cat.png")

print(
```

```
    f"Max memory allocated: {bytes_to_giga_bytes(torch.cuda.
        max_memory_allocated())} GB"
)
```

Listing A.2: Memory-efficient image generation with PixArt-XL (8-bit encoder and offloading). Source URL: https://gist.github.com/sayakpaul/3ae0f847001d342af27018a96f467e4e

A.1.3. pixart_batch_512.py

```
# pip install -U accelerate transformers bitsandbytes
# pip install -U git+https://github.com/huggingface/diffusers

from transformers import T5EncoderModel
from diffusers import PixArtAlphaPipeline
import torch
import gc
from PIL import Image
import os

def flush():
    gc.collect()
    torch.cuda.empty_cache()

def bytes_to_giga_bytes(bytes):
    return bytes / 1024 / 1024 / 1024

# Load text encoder (in 8-bit for memory efficiency)
text_encoder = T5EncoderModel.from_pretrained(
    "PixArt-alpha/PixArt-XL-2-512x512",
    subfolder="text_encoder",
    load_in_8bit=True,
    device_map="balanced",
)

pipe = PixArtAlphaPipeline.from_pretrained(
    "PixArt-alpha/PixArt-XL-2-512x512",
```

```
    text_encoder=text_encoder,
    transformer=None,
    device_map="balanced"
)

# Encode prompt once (empty string in this case)
with torch.no_grad():
    prompt = ""
    prompt_embeds, prompt_attention_mask, negative_embeds,
        negative_prompt_attention_mask = pipe.encode_prompt(prompt)

# Free memory from encoder
del text_encoder
del pipe
flush()

# Reload full pipeline without encoder
pipe = PixArtAlphaPipeline.from_pretrained(
    "PixArt-alpha/PixArt-XL-2-512x512",
    text_encoder=None,
    torch_dtype=torch.float16,
).to("cuda")

# Generate 10 200 images
for i in range(1, 10_201):
    with torch.no_grad():
        latents = pipe(
            negative_prompt=None,
            prompt_embeds=prompt_embeds,
            negative_prompt_embeds=negative_embeds,
            prompt_attention_mask=prompt_attention_mask,
            negative_prompt_attention_mask=negative_prompt_attention_mask,
            num_images_per_prompt=1,
            output_type="latent",
        ).images

```

```
        # Decode latent to image
        image = pipe.vae.decode(
            latents / pipe.vae.config.scaling_factor,
            return_dict=False
        )[0]
        image = pipe.image_processor.postprocess(image, output_type="pil")[0]

        # Save image with padded filename
        filename = f"image_{i:05d}.png"
        image.save(filename)

        print(f"Saved: {filename}")

        # Clean up between iterations
        flush()

print(
    f"Max memory allocated: "
    f"{bytes_to_giga_bytes(torch.cuda.max_memory_allocated()):.2f} GB"
)
```

Listing A.3: Storage-efficient generation of 10,200 PixArt XL images (512x512)

[2]

A.1.4. sd15_generator.py

```
from diffusers import DiffusionPipeline

# Load the model
model_id = "stable-diffusion-v1-5/stable-diffusion-v1-5"
pipeline = DiffusionPipeline.from_pretrained(
    model_id,
    use_safetensors=True
)

```

[2] Created with help from ChatGPT o3 and ChatGPT 4o

```python
# Set up the prompt and device
prompt = ""
pipeline = pipeline.to("cuda")

# Loop to generate 10 000 images
for i in range(1, 10_001):
    # Generate the image
    image = pipeline(prompt).images[0]

    # Save the image with a count-based filename
    image.save(f"image_{i}.png")

    print(f"Saved image_{i}.png")
```

Listing A.4: Image generation with Stable Diffusion 1.5 (10,000 samples)

3

A.1.5. playground_generator.py

```python
from diffusers import DiffusionPipeline
import torch

pipe = DiffusionPipeline.from_pretrained(
    "playgroundai/playground-v2-512px-base",
    torch_dtype=torch.float16,
    use_safetensors=True,
    variant="fp16",
)
pipe.to("cuda")

prompt = ""
for i in range(1, 10_401):
    # Generate the image
    image = pipe(prompt=prompt, width=512, height=512).images[0]

```

[3] Created with help from ChatGPT o3 and ChatGPT 4o

```
    # Save the image with a count-based filename
    image.save(f"image_{i}.png")

    print(f"Saved image_{i}.png")
```

Listing A.5: Image generation with `playground-v2` (10,400 samples, 512x512)

4

A.2. Deletion/Cleaning scripts

A.2.1. png_cleaner.py

```
import os

# Use the current working directory
folder_path = os.getcwd()

# Minimum size in bytes (1 KB)
min_size_bytes = 1024

# Counter for deleted images
deleted_count = 0

# Process each PNG file in the current directory
for filename in os.listdir(folder_path):
    if filename.lower().endswith(".png"):
        file_path = os.path.join(folder_path, filename)
        try:
            file_size = os.path.getsize(file_path)
            if file_size < min_size_bytes:
                os.remove(file_path)
                print(f"Deleted: {filename} ({file_size} bytes)")
                deleted_count += 1
        except Exception as e:
```

[4] Created with help from ChatGPT o3 and ChatGPT 4o

```
            print(f"Error checking file {filename}: {e}")

print(f"Total deleted images: {deleted_count}")
```

Listing A.6: Python script for removing blacked-out images after generation

5

A.3. Code of parsers for JPEG compression

A.3.1. png2jpg.py

```
#!/usr/bin/env python3
"""
png2jpg.py - Convert all PNG images in INPUT_DIR to compressed JPEGs in
    OUTPUT_DIR.

Usage:
    python png2jpg.py /path/to/input_pngs /path/to/output_jpegs [-q 85]

Requires:
    pip install pillow
"""

import argparse
from pathlib import Path
from PIL import Image
import sys

def convert_png_to_jpg(src_dir: Path, dst_dir: Path, quality: int) -> None:
    """Convert all PNG files in src_dir to JPEGs in dst_dir."""
    png_files = list(src_dir.glob("*.png"))
    if not png_files:
        print(f"[WARN] No PNG files found in {src_dir}")
        return

```

[5] Created with help from ChatGPT o3 and ChatGPT 4o

```
24      dst_dir.mkdir(parents=True, exist_ok=True)
25
26      for png_path in png_files:
27          try:
28              with Image.open(png_path) as im:
29                  rgb = im.convert("RGB")         \# JPEG cant store transparency
30                  jpg_path = dst_dir / (png_path.stem + ".jpg")
31                  rgb.save(jpg_path, "JPEG", quality=quality, optimize=True)
32                  print(f"[OK] {png_path.name}  ->  {jpg_path.name}")
33          except Exception as e:
34              print(f"[ERR] {png_path.name}: {e}")
35
36  def self_check(src_dir: Path, dst_dir: Path) -> bool:
37      """Return True iff every PNG now has a corresponding non-empty JPEG."""
38      png_stems = {p.stem for p in src_dir.glob("*.png")}
39      jpg_files  = {p for p in dst_dir.glob("*.jpg") if p.stat().st_size > 0}
40      jpg_stems  = {p.stem for p in jpg_files}
41
42      missing = png_stems - jpg_stems
43      extras  = jpg_stems - png_stems   \# shouldnt happen, but we check anyway
44
45      if missing:
46          print(f"[FAIL] Missing {len(missing)} JPEG(s): {', '.join(sorted(
                missing))}")
47      if extras:
48          print(f"[WARN] Extra JPEG(s) with no source PNG: {', '.join(sorted(
                extras))}")
49
50      if not missing:
51          print(f"[SUCCESS] Converted {len(jpg_files)} image(s). All good")
52      return not missing
53
54  def main() -> None:
55      parser = argparse.ArgumentParser(description="Convert PNGs to JPEGs with a
            self-check.")
```

```
56      parser.add_argument("input_dir",  type=Path, help="Folder containing PNG
            images")
57      parser.add_argument("output_dir", type=Path, help="Folder to write JPEG
            images")
58      parser.add_argument("-q", "--quality", type=int, default=85,
59                          help="JPEG quality (1-95, default 85)")
60      args = parser.parse_args()
61
62      if not args.input_dir.is_dir():
63          sys.exit(f"[ERROR] Input directory {args.input_dir} does not exist or
                is not a directory.")
64
65      convert_png_to_jpg(args.input_dir, args.output_dir, args.quality)
66      ok = self_check(args.input_dir, args.output_dir)
67      sys.exit(0 if ok else 1)
68
69  if __name__ == "__main__":
70      main()
```

Listing A.7: PNG to JPEG converter with self-check

[6]

A.3.2. tif2jpg.py

```
1   #!/usr/bin/env python3
2   """
3   tif2jpg.py - Convert all TIF/TIFF images in INPUT_DIR to compressed JPEGs in
        OUTPUT_DIR.
4
5   Usage
6   -----
7       python tif2jpg.py /path/to/tifs /path/to/jpegs [-q 85]
8
9   Dependencies
10  ------------
```

[6] Created with help from ChatGPT o3 and ChatGPT 4o

```
    pip install pillow
"""

import argparse
import sys
from pathlib import Path
from PIL import Image


def list_tif_files(directory: Path):
    """Return a list of .tif / .tiff files (case-insensitive) in *directory*."""
    return [
        p for ext in ("*.tif", "*.tiff", "*.TIF", "*.TIFF")
        for p in directory.glob(ext)
    ]


def convert_tif_to_jpg(src_dir: Path, dst_dir: Path, quality: int) -> None:
    """Convert every TIF/TIFF in *src_dir* to a JPEG in *dst_dir*."""
    tif_files = list_tif_files(src_dir)
    if not tif_files:
        print(f"[WARN] No TIF/TIFF files found in {src_dir}")
        return

    dst_dir.mkdir(parents=True, exist_ok=True)

    for tif_path in tif_files:
        try:
            with Image.open(tif_path) as im:
                rgb = im.convert("RGB")              # drop alpha for JPEG
                jpg_path = dst_dir / (tif_path.stem + ".jpg")
                rgb.save(jpg_path, "JPEG", quality=quality, optimize=True)
                print(f"[OK] {tif_path.name} -> {jpg_path.name}")
        except Exception as exc:
            print(f"[ERR] {tif_path.name}: {exc}")
```

```
46
47
48  def self_check(src_dir: Path, dst_dir: Path) -> bool:
49      """Ensure every source TIF/TIFF has a non-empty JPEG counterpart."""
50      src_stems = {p.stem for p in list_tif_files(src_dir)}
51      jpg_stems = {p.stem for p in dst_dir.glob("*.jpg") if p.stat().st_size >
            0}
52
53      missing = src_stems - jpg_stems
54      extras  = jpg_stems - src_stems
55
56      if missing:
57          print(f"[FAIL] Missing {len(missing)} JPEG(s): {', '.join(sorted(
                missing))}")
58      if extras:
59          print(f"[WARN] Extra JPEG(s) with no source TIF: {', '.join(sorted(
                extras))}")
60
61      if not missing:
62          print(f"[SUCCESS] Converted {len(jpg_stems)} image(s). All good")
63      return not missing
64
65
66  def main():
67      parser = argparse.ArgumentParser(
68          description="Convert TIF/TIFF images to JPEG with a post-run self-
                check."
69      )
70      parser.add_argument("input_dir",  type=Path, help="Folder containing TIF/
            TIFF images")
71      parser.add_argument("output_dir", type=Path, help="Folder to write JPEG
            images")
72      parser.add_argument(
73          "-q", "--quality", type=int, default=85,
74          help="JPEG quality (1-95, default 85)"
75      )
```

```
76      args = parser.parse_args()
77
78      if not args.input_dir.is_dir():
79          sys.exit(f"[ERROR] Input directory {args.input_dir} does not exist or
                is not a directory.")
80      if not 1 <= args.quality <= 95:
81          sys.exit("[ERROR] Quality must be between 1 and 95.")
82
83      convert_tif_to_jpg(args.input_dir, args.output_dir, args.quality)
84      ok = self_check(args.input_dir, args.output_dir)
85      sys.exit(0 if ok else 1)
86
87
88  if __name__ == "__main__":
89      main()
```

Listing A.8: Conversion from TIF/TIFF to JPEG with subsequent self-check

[7]

A.4. Randomizer scripts

A.4.1. random_image_subset.py

```
1   #!/usr/bin/env python3
2   """
3   random_image_subset.py - Copy a random subset of images, without ever
4   choosing the same image twice (even across multiple runs).
5
6   Usage
7   -----
8       python random_image_subset.py SRC_DIR DST_DIR [-n 10000]
9                                                   [--extensions .jpg .png]
10
11  Dependencies: none beyond Python >=3.8
12  """
```

[7] Created with help from ChatGPT o3 and ChatGPT 4o

```
13
14 import argparse
15 import random
16 import shutil
17 import sys
18 from pathlib import Path
19 from typing import List, Set
20
21
22 def list_images(folder: Path, exts: List[str]) -> List[Path]:
23     """List image files in *folder* whose suffix matches *exts* (case-
       insensitive)."""
24     allowed = {e.lower() for e in exts}
25     return [p for p in folder.iterdir()
26             if p.is_file() and p.suffix.lower() in allowed]
27
28
29 def existing_basenames(folder: Path) -> Set[str]:
30     """Return a set of basenames already present in *folder* (without
       extension)."""
31     return {p.stem for p in folder.iterdir() if p.is_file()}
32
33
34 def choose_unique(src_files: List[Path], seen_stems: Set[str], n: int) -> List
    [Path]:
35     """
36     Pick up to *n* files whose stem is *not* in seen_stems.
37     Ensures no duplicate stems in the result.
38     """
39     unseen = [p for p in src_files if p.stem not in seen_stems]
40     if not unseen:
41         return []
42     sample_size = min(n, len(unseen))
43     return random.sample(unseen, sample_size)
44
45
```

```
46 def main():
47     parser = argparse.ArgumentParser(
48         description="Copy N random images from SRC_DIR to DST_DIR "
49                     "without ever repeating the same basename."
50     )
51     parser.add_argument("src_dir", type=Path, help="Source directory with
           images")
52     parser.add_argument("dst_dir", type=Path, help="Destination directory")
53     parser.add_argument("-n", "--number", type=int, default=10_000,
54                         help="How many images to copy (default 10 000)")
55     parser.add_argument("--extensions", nargs="+", default=[
56                         ".jpg", ".jpeg", ".png", ".tif", ".tiff", ".bmp", ".
                               gif"],
57                         help="Which file extensions count as images")
58     args = parser.parse_args()
59
60     if not args.src_dir.is_dir():
61         sys.exit(f"[ERROR] {args.src_dir} is not a directory.")
62     if args.number < 1:
63         sys.exit("[ERROR] --number must be at least 1.")
64
65     # Collect candidates and already-copied basenames
66     candidates = list_images(args.src_dir, args.extensions)
67     args.dst_dir.mkdir(parents=True, exist_ok=True)
68     already_there = existing_basenames(args.dst_dir)
69
70     picked = choose_unique(candidates, already_there, args.number)
71
72     if not picked:
73         print("[INFO] No new unique images to copy.")
74         sys.exit(1)
75
76     for src in picked:
77         shutil.copy2(src, args.dst_dir / src.name)
78         print(f"[COPY] {src.name}")
79
```

```
    print(f"[DONE] Copied {len(picked)} new image(s) -> {args.dst_dir}")
    sys.exit(0)


if __name__ == "__main__":
    main()
```

Listing A.9: Copy random, unique subset of images

8

A.5. Renaming scripts

A.5.1. rename_script.py

Listing A.10: Rename and copy files script

```
    import os
    import shutil
    import argparse

    # Define a function that renames and copies files
    def copy_files(source_folder, destination_folder, custom_name):
        # Ensure destination folder exists
        if not os.path.exists(destination_folder):
            os.makedirs(destination_folder)

        # Loop through the files in the source folder
        for filename in os.listdir(source_folder):
            # Get the full path of the file
            file_path = os.path.join(source_folder, filename)

            # Check if it's a file (skip directories)
            if os.path.isfile(file_path):
                # Split the file name and extension
                name, extension = os.path.splitext(filename)
```

[8] Created with help from ChatGPT o3 and ChatGPT 4o

```
20
21                  # Create the new name by adding the custom name before the
                        extension
22                  new_name = f"{name}_{custom_name}{extension}"
23
24                  # Get the full path for the new file
25                  new_file_path = os.path.join(destination_folder, new_name)
26
27                  # Copy the file to the new location
28                  shutil.copy(file_path, new_file_path)
29                  print(f"Copied and renamed: {filename} -> {new_name}")
30
31      # Set up argument parsing
32      def main():
33          parser = argparse.ArgumentParser(description="Rename and copy files
                from one folder to another.")
34          parser.add_argument("source_folder", help="The path to the source
                folder containing the files.")
35          parser.add_argument("destination_folder", help="The path to the
                destination folder.")
36          parser.add_argument("custom_name", help="The custom name to add to the
                 file before the extension.")
37
38          args = parser.parse_args()
39
40          # Call the function to rename and copy files
41          copy_files(args.source_folder, args.destination_folder, args.
                custom_name)
42
43      if __name__ == "__main__":
44          main()
```

9

[9] Created with help from ChatGPT o3 and ChatGPT 4o

A.6. Data science

A.6.1. ter_opt.py

```
#!/usr/bin/env python3
"""
ter_opt.py  -  Bestimmt den optimalen Schwellenwert \tau* und die minimale
Total Error Rate (TER) aus einer CSV-Datei mit den Spalten  score , label .

Aufruf:
    python ter_opt.py path/to/scores_labels.csv
"""

import argparse
import pandas as pd
import numpy as np
import sys

def ter_min_from_csv(csv_path: str):
    # Einlesen und Validieren
    df = pd.read_csv(csv_path)
    if not {"score", "label"}.issubset(df.columns):
        sys.exit("CSV muss die Spalten  score  und  label  enthalten.")
    if df["label"].nunique() != 2 or not set(df["label"].unique()) <= {0,
        1}:
        sys.exit("Spalte  label  muss nur die Werte 0 (Cover) und 1 (Stego)
            enthalten.")

    # Sortieren nach Score
    df = df.sort_values("score").reset_index(drop=True)

    # Gesamtanzahlen
    N_cover = (df["label"] == 0).sum()
    N_stego  = (df["label"] == 1).sum()

    # Kumulative Summen
    df["cum_stego"] = (df["label"] == 1).cumsum()
```

```
32      df["cum_cover"] = (df["label"] == 0).cumsum()
33
34      # Fehlerraten & TER
35      df["FNR"] = df["cum_stego"] / N_stego
36      df["FPR"] = (N_cover - df["cum_cover"]) / N_cover
37      df["TER"] = 0.5 * (df["FNR"] + df["FPR"])
38
39      # Minimum finden
40      idx_min   = df["TER"].idxmin()
41      tau_star  = df.at[idx_min, "score"]
42      ter_min   = df.at[idx_min, "TER"]
43
44      return tau_star, ter_min
45
46  def main():
47      parser = argparse.ArgumentParser(description="Optimalen \tau und
            minimale TER berechnen")
48      parser.add_argument("csv", help="CSV-Datei mit Spalten score,label")
49      args = parser.parse_args()
50
51      tau_star, ter_min = ter_min_from_csv(args.csv)
52      print(f"Optimaler Schwellenwert \tau*: {tau_star:.6f}")
53      print(f"Minimale Total Error Rate:  {ter_min:.4%}")
54
55  if __name__ == "__main__":
56      main()
```

Listing A.11: Determination of the optimal threshold value τ^* and the minimum TER

10

[10] Created with help from ChatGPT o3 and ChatGPT 4o