Implementation and Benchmarking of Perceptual Image Hash Functions

Christoph Zauner

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Sichere Informationssysteme

in Hagenberg

im Juli 2010

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 14. Juli 2010

Christoph Zauner

Acknowledgements

I would like to begin by thanking my advisors. Both Prof. Eckehard Hermann, from the University of Applied Sciences Hagenberg[1] in Austria and Dr. Martin Steinebach, from the Fraunhofer Institute for Secure Information Technology[2] in Germany were valuable resources of knowledge and ideas.

Furthermore, I would like to thank my parents. Their unconditional support is largely the reason that I was able to write this thesis. I am also grateful to my girlfriend Marlene who has kept exemplary patience while I completed my thesis.

Finally, I would like to thank my dear friend Daniel. Without his help I would have not been able to get access to most of the scientific papers I relied upon.

[1] Homepage: `http://www.fh-ooe.at/en/upper-austria/`

[2] Homepage: `http://www.sit.fraunhofer.de/EN/`

Contents

List of Figures

List of Listings

List of Tables

Abstract

Perceptual image hash functions produce hash values based on the image's visual appearance. A perceptual hash can also be referred to as e.g. a robust hash or a fingerprint. Such a function calculates similar hash values for similar images, whereas for dissimilar images dissimilar hash values are calculated. Finally, using an adequate distance or similarity function to compare two perceptual hash values, it can be decided whether two images are perceptually different or not. Perceptual image hash functions can be used e.g. for the identification or integrity verification of images.

This thesis proposes a novel benchmarking framework, called Rihamark, for perceptual image hash functions. Subsequently, four different perceptual image hash functions were benchmarked: A discrete Cosine transform (DCT) based , a Marr-Hildreth operator based, a radial variance based and a block mean value based image hash function. pHash, an open source implementation of various perceptual hash functions, was used to benchmark the first three functions. The latter, the block mean value based image hash function was implemented by the author of this thesis himself.

The block mean value based image hash function outperforms the other hash functions in terms of speed. The DCT based image hash function is the slowest. Although the Marr-Hildreth operator based image hash function is not the fastest nor the most robust, it offers by far the best discriminiative abilities. Interestingly enough, the performance in terms of discriminiative ability does not depend on the content of the images. That is, no matter whether the visual appearance of the images compared was very similar or not, the performance of the particular hash function did not change significantly. Different image operations, like horizontal flipping, rotating or resizing, were used to test the robustness of the image hash functions. An interesting result is that none of the tested image hash function is robust against flipping an image horizontally.

Kurzfassung

Wahrnehmungsbasierte Hashfunktionen für Bilder produzieren Hashwerte die von der visuellen Wahrnehmung eines Bildes abhängen. Andere gebräuchliche Bezeichnungen sind zum Beispiel robuste Hashes oder Fingerprints. Solch eine Hashfunktion berechnet ähnliche Hashwerte für ähnliche Bilder, wohingegen für unterschiedliche Bilder unterschiedliche Hashwerte berechnet werden. Letztenendes kann mit Hilfe einer passenden Distanz- oder Ähnlichkeitsfunktion entschieden werden ob zwei Bilder wahrnehmbar verschieden sind oder nicht. Wahrnehmungsbasierte Hashfunktionen für Bilder werden unter anderem zur Identifikation oder zur Verifikation der Integrität eingesetzt.

Diese Diplomarbeit stellt ein neuartiges Benchmarking Framework, genannt Rihamark, für wahrnehmungsbasierte Bildhashfunktionen vor. In weiterer Folgen wurden vier unterschiedliche wahrnehmungsbasierte Hashfunktionen für Bilder getestet: Eine discrete Cosine transform (DCT) basierte, eine Marr-Hildreth Operator basierte, eine auf der "radialen Varianz" basierende und eine auf Mittelwerten von Blöcken basierende Hashfunktion für Bilder. pHash, eine Open Source Implementierung von verschiedenen wahrnehmungsbasierten Hashfunktionen wurde für die Tests der ersten drei Funktionen benutzt. Die zuletzt genannte Funktion, die Hashfunktion basierend auf den Mittelwerten von Blöcken, wurde vom Author dieser Arbeit selbst implementiert.

Die auf den Mittelwerten von Blöcken basierende Hashfunktion ist die schnellste. Die DCT basierte ist die langsamste. Obwohl die Hashfunktion basierend auf dem Marr-Hildreth Operator weder die schnellste, noch die robusteste ist, ist sie mit Abstand die Beste in Bezug auf die Unterscheidungsfähigkeit von Bildern. Interessanterweise wird die Unterscheidungsfähigkeit der getesteten Hashfunktionen nicht durch den Inhalt der Bilder beeinflusst. Das soll heißen, egal ob sich die visuellen Wahrnehmungen der verwendeten Bilder glichen oder nicht, die Performanz der jeweiligen Hashfunktion änderte sich nur unbeträchtlich. Unterschiedliche Bildoperationen, wie horizontal Spiegeln, Rotieren oder Ändern der Größe wurden verwendet um die Robustheit der Hashfunktionen zu testen. Ein interessantes Ergebniss ist, dass keine der getesteten Hashfunktionen robust gegen das horizontale Spiegeln eines Bildes ist.

Chapter 1

Introduction

1.1 Motivation and Purpose of Thesis

Due to the ever increasing digitalization, the authentication of multimedia content is becoming more and more important. Authentication in general means deciding whether an object is authentic or not. That is, if it matches a given original object. The authentication depends heavily on the type of the object. When authenticating an executable file, it is important that every single bit exactly matches the original executable. Cryptographic hash functions are adequate for such tasks. To check the authenticity of multimedia content, other methods are better suited. A multimedia object, e.g. an image, can have different digital representations that all look the same to the human perception. Different digital representations can emerge from an image through image processing steps like cropping, compression or histogram equalization. Each of these image processing steps changes the binary representation of the image. Using a cryptographic hash function to authenticate the modified images therefore does not work.

So-called perceptual hash functions have been proposed to establish the "perceptual equality" of multimedia content. In recent years, a growing scientific and industrial interest in perceptual hashing technology has been seen. Such functions have been developed for different digital media types (e.g. audio, image or video). Perceptual hash functions extract certain features from multimedia content and calculate a hash value based on these features. When authenticating a multimedia object the hash values of the original object and the object to be authenticated are compared using specific functions. Such functions calculate a distance or similarity score between two perceptual hash values. The final verdict is based on a chosen threshold.

The purpose of this thesis is to discuss the benchmarking and implementation of previously published perceptual hash functions for images. A lot of scientific work has been done in this area but ready-to-use benchmarking solutions for such functions or implementations of perceptual hash functions for images are still not available. Facing so many different perceptual image hash functions and applications, a readily available benchmark tool for perceptual image hashing is desirable. A decision maker needs a benchmark tool to help him to compare different functions for a given application scenario. Developers of functions also need such a tool to determine the best application scenarios for their functions, investigate the limits of their functions, or find out how to improve their functions by adjusting their parameters and comparing the corresponding results.

1.2 Terms Related to Perceptual Hashing

Perceptual hash functions are an interdisciplinary field of research. Cryptography, digital watermarking and digital signal processing are part of this field of research. Hence there is no uniform or consistent nomenclature. A perceptual hash can also be referred to as

- a fingerprint,
- a passive fingerprint,
- a perceptual checksum,
- a robust hash,
- or a soft hash.

The term passive fingerprint is used because the multimedia content itself is not changed, whereas active fingerprints change the content. Digital watermarking algorithms can be used to embed a fingerprint directly into a multimedia object. Such fingerprints are active fingerprints for instance.

In this thesis the generic term “**media object**” is used for multimedia content such as audio, image or video files. A media object can be changed using various **operations**. An example of an image operation is cropping image by 10%. Operation is a generic term for modification or manipulation.

Definition 1.1 (Modification):
A modification is defined as an operation that does not alter the essential content of a media object.

After a modification, a media object is still expected to be detected as authentic by a perceptual hash function.

Definition 1.2 (Manipulation):
A manipulation is defined as an operation that does alter the essential content of a media object.

After a manipulation, a media object is expected to be detected as not authentic (inauthentic) by a perceptual hash function.

Chapter 2

Review of Perceptual Hashing

In this chapter basic concepts and terms related to perceptual hash functions are discussed. Furthermore, related topics, namely cryptographic hash functions and digital watermarking are reviewed and similarities are identified.

2.1 Perceptual Hash Functions

To ease the understanding of hash functions and the relationship of perceptual hash functions to e.g. cryptographic hash functions, the general definition of a hash function will be discussed first. At the highest level, hash functions can be categorized into **unkeyed hash functions** and **keyed hash functions**.[27, p. 322] An unkeyed hash function H generates a hash value[1] h from an arbitrary input x (that is $h = H(x)$). A keyed hash function generates a hash value h from an arbitrary input x and a secret key k (that is $h = H(x, k)$). Keyed hash functions are also called Message Authentication Codes (MACs). We restrict our attention to unkeyed hash functions. [27, p. 322] defines them as follows.

Definition 2.1 (Unkeyed hash function):
A hash function is [. . .] a function H which has, as a minimum, the following two properties:

- *compression – H maps an input x of arbitrary finite bit length, to an output $H(x)$ of fixed bit length n.*
- *ease of computation – given H and an input x, $H(x)$ is easy to compute.*

[1] Also referred to as hash code, hash result, or simply hash.

The rest of section 2.1 will focus on the discussion of perceptual hash functions. According to [26, sec. 1], a perceptual hash function should possess four properties (also see [28, sec. 2] and [32, sec. 2.2]). Let P denote probability. Let H denote a hash function which takes one media object (e.g an image) as input and produces a binary string of length l. Let x denote a particular media object and $\hat{x}$ denote a modified version of this media object which is “perceptually similar” to x. Let y denote a media object that is “perceptually different” from x. Let x' and y' denote hash values. $\{0/1\}^l$ represents binary strings of length l. Then the four desirable properties of a perceptual hash are identified as follows.

- Equal distribution (unpredictability) of hash values:

$$P(H(x) = x') \approx \frac{1}{2^l}, \forall x' \in \{0/1\}^l \tag{2.1}$$

- Pairwise independence for perceptually different media objects x and y:

$$P(H(x) = x' | H(y) = y') \approx P(H(x) = x'), \forall x', y' \in \{0/1\}^l \tag{2.2}$$

- Invariance for perceptually similar media objects x and $\hat{x}$:

$$P(H(x) = H(\hat{x})) \approx 1 \tag{2.3}$$

- Distinction of perceptually different media objects x and y:

$$P(H(x) = H(y)) \approx 0 \tag{2.4}$$

To meet property (equation) 2.3, most perceptual hash functions try to extract features of media objects which are invariant under insignificant global modifications.[28, sec. 1] For images, such global modifications are compression or cropping, for instance. Property 2.4 also means that, given a media object x, it should be nearly impossible to construct a perceptually different media object y such that $H(x) = H(y)$. Because the features used by published perceptual hash functions are publicly known, this property can be especially hard to achieve.[28, sec. 1]

The properties conflict with each other.[32, sec. 2.2] When **identifying** media objects a small number of false positives does not encumber the system. Instead, it is much more important that for any media object x it is impossible to construct a perceptually similar media object $\hat{x}$ such that $H(x) \neq H(\hat{x})$. This also includes the creation of $\hat{x}$ from x through any kind of operation (e.g. cropping in the case of an image).[28, sec. 1] When using a perceptual hash functions to **authenticate** media objects, even a small number of false positives is unacceptable. For an adversary, it must be impossible for any media object x to construct a perceptually different media object y such that $H(x) = y$. Thus, property 2.3 will have to be neglected in favour of property 2.4. Likewise for perfect unpredictability, a equal distribution of the hash values is needed. This would deter achieving the property 2.3.[32, sec. 2.2] Depending on the application, perceptual hash functions have to achieve these (conflicting) properties to some extent and/or facilitate trade-offs.[32, sec. 2.2] So the necessity for a perceptual hash function to exhibit each one of these properties changes slightly depending on the application in which such an algorithm is used.

A problem when developing perceptual hash functions is that authentic media objects can not be precisely separated from not authentic ones. To get a better understanding of this problem the following example is given. The Joint Photographic Experts Group (JPEG) compression is an image operation which normally does not change an image in a perceptually significant way. That is, applying JPEG compression to an image should not render it inauthentic. Nonetheless JPEG compression, especially when applied using low quality settings, can blur an image significantly. Therefore especially images which contain small details that are important to their semantic meaning (e.g. an image which comprises road signs and car number plates) can be affected severely when JPEG compression is applied and thus should be recognized as not authentic. [5] summarizes this as follows:

> *"For some processing operations it is difficult to decide if the result of the modifications is authentic. In addition to perceptive issues this decision boundary is influenced by the application scenarios."*

Therefore [37] proposes a continuous interpretation of authentic:

> *"An image which is bit by bit identical to the original image is considered completely authentic (authenticity measure of 1.0). An image which has nothing in common with the original image would be considered not authentic (authenticity measure of 0.0). All other images would be partially authentic. Partially authentic is a loosely defined concept and measurement of the authenticity is subjective, and changes from application domain to application domain."*

This authenticity measure can be illustrated as an autheticity vs. modification curve. For each different type of modification there would be a corresponding curve. Figure 2.1 illustrates an example of such an authenticity vs. modification curve. The "JPEG Compression" curve relates the

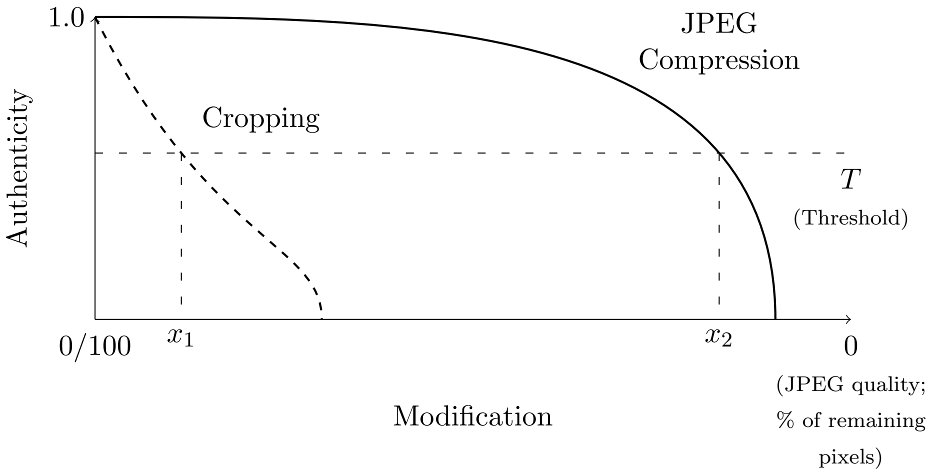


Figure 2.1: Authenticity vs. Modification Curve. Cp. [37].

authenticity of an image to the quality factor of the used JPEG compression. The "Cropping" curve relates the authenticity to the percentage of the remaining pixels after the image has been cropped. Using a threshold, the authenticity could be measured in a binary quantity – as it is common in cryptography. According to figure 2.1, cropping an image by 100 to $100 - x_1$ percents would render an image inauthentic. The same is true for the case that the image is compressed using a JPEG quality setting between x_2 and 0. Thus the authenticity vs. modification curve should have a gentle slope for modifications and a steep one for manipulations.

2.1.1 Usage Modes

Actual applications of perceptual hash functions are image spam detection, searching the internet for copyright violations or maintaining databases of illegal content such as child pornography. Forensic programs like EnCase[2] or the Forensic Toolkit[3] only use cryptographic hash functions to index and search files. Perceptual hash functions would be a reasonable addition to those programs. Despite different application scenarios which deploy perceptual hash functions, various common "usage modes" can be derived.[6, sec. 2.1.2] The usage modes are as follows:

1. Content identification

2. Integrity verification

3. Watermarking support

[2] Homepage: `http://www.guidancesoftware.com/`

[3] Homepage: `http://www.accessdata.com/forensictoolkit.html`

4. Content-based media retrieval and processing

This section gives a brief overview of each of the aforementioned usage modes.

Content Identification

Perceptual hash functions can offer excellent performance when searching large databases for desired multimedia content. For instance, [18] proposes a perceptual audio hash function and a very efficient search strategy which enable searching a large perceptual audio hash database efficiently. Using perceptual hash functions for such applications also means that only the hash values and the corresponding meta data (e.g. file name) need to be stored in the database. There is no need to store the multimedia objects themselves in the database. This reduces the size of the database dramatically. And of course, another advantage is that if the media object has been modified in a perceptually insignificant way, it still can be found in the database. As previously discussed in section 2.1 a perceptual hash function optimized for this usage mode will have to neglect property (equation) 2.4 in favour of 2.3.

Figure 2.2 illustrates this usage mode. It is divided into two phases. The "database creation" and the "content identification" phase. During the database creation phase, the database is filled with perceptual hash values of media objects that should be recognizable later on. Usually, additional meta data of each media object is stored with its hash value. This can be e.g. the file name of a media object, its ID3 tag, if it is an audio file, or its Exchangeable image file format (Exif) tags if it is an image file. In the content identification phase, an unidentified media object is presented to the system. The media object is processed in order to obtain a perceptual hash. The perceptual hash is then compared with the hash values stored in the database. If there is a match, the system will provide further information about the beforehand unidentified media object (available meta data, reliability measure of the match, . . .).

Figure 2.3 illustrates the "perceptual hash extraction" and the "matching" function of the content identification phase more detailed. The database has already been populated during the "database population" phase. It now includes the perceptual hashes and corresponding meta data of media objects. The procedure is as follows:

Feature extraction and processing: Normally the media content must be preprocessed in order to be processed by a perceptual hash function. In the case of an image, such required preprocessing steps can be to resize the image to a given resolution or to convert it to levels of grey. Hereafter, features which are needed for the modelling of the perceptual hash are extracted from the media content.

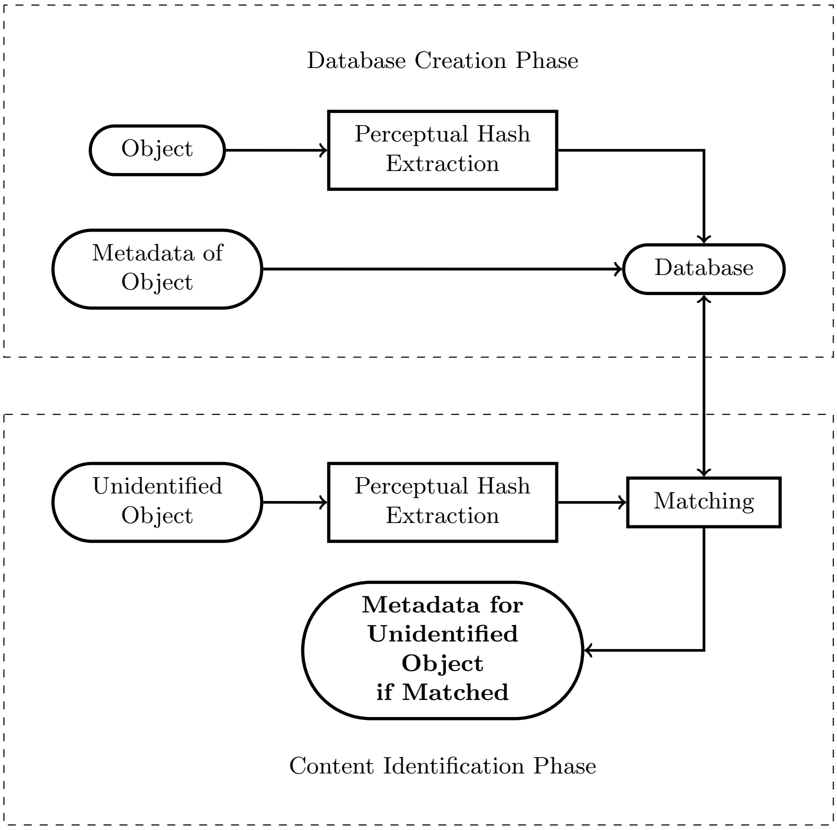

Database Creation Phase
Object
Perceptual Hash Extraction
Metadata of Object
Database
Unidentified Object
Perceptual Hash Extraction
Matching
Metadata for Unidentified Object if Matched
Content Identification Phase

Figure 2.2: Usage mode “identification”. Cp. [6].

Modelling of perceptual hash: A perceptual hash is calculated using the features extracted in the previous step.

Database look-up: To compare two perceptual hashes, special search algorithms (e.g. [30]) and distance/similarity functions according to the used perceptual hash function must be used. Various distance and similarity functions are discussed in section 2.1.2.

Hypothesis testing: Based on a pre-defined threshold it is determined if there is a match. Therefore the determination of an adequate threshold, in accordance with the actual application scenario, is critical.

Integrity Verification

Basically, perceptual hash functions can be used in two different ways to verify the integrity of a media object.[37] The common architecture is illustrated in Figure 2.4. On the one hand perceptual hashes can be embedded

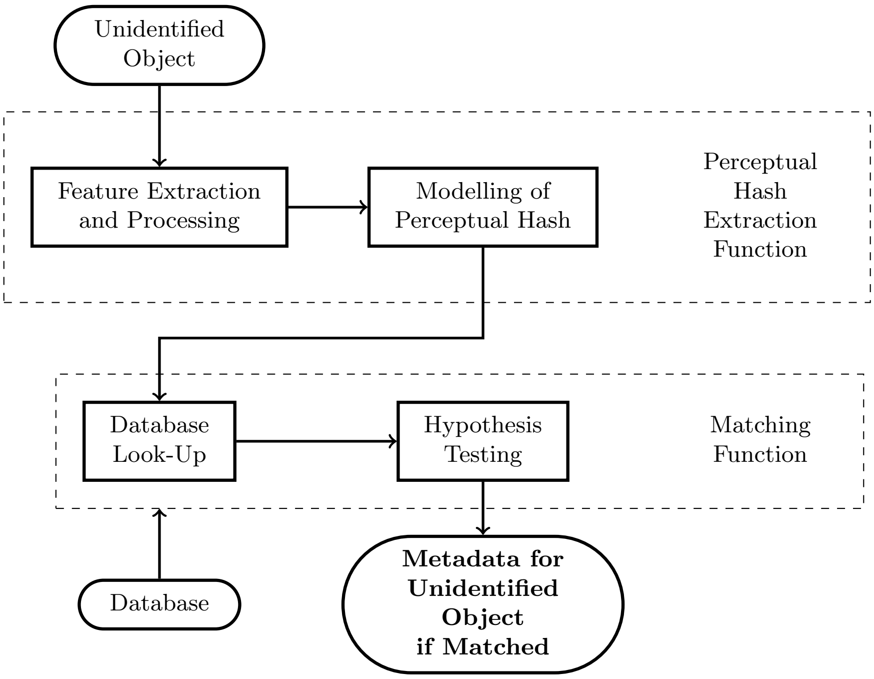

Unidentified Object
Feature Extraction and Processing
Modelling of Perceptual Hash
Perceptual Hash Extraction Function
Database Look-Up
Hypothesis Testing
Matching Function
Database
Metadata for Unidentified Object if Matched

Figure 2.3: Detailed look at the "perceptual hash extraction" function and the "matching" function during the "content identification" phase. Cp. [6].

directly in the multimedia content using digital watermarks. This is further described in the section about the watermarking support usage mode. On the other hand a digital signature[27, ch. 11] can be used to sign the perceptual hash. Figure 2.5 illustrates the creation of such a digital signature, whereas figure 2.6 shows its verification. Beyond that, some perceptual hash functions (e.g. [36]) are able to report the type of manipulation and where in the multimedia object it occured. As previously discussed in section 2.1 a perceptual hash function optimized for this usage mode will have to neglect property (equation) 2.3 in favour of 2.4.

A digital signature can be used for more than just image authentication. Together with a secure timestamp it can be used as a proof of first authorship. A watermark allows for verification of the origin of a media object. However, a digital watermark alone is unsuitable to prove first authorship, because a media object could be marked with multiple digital watermarks.[37] Furthermore, digital watermarks are not adequate for protecting the authenticity of media objects.[25, sec. 6] In general a digital signature protects the receiver of a media object, whereas a digital watermark protects the author.

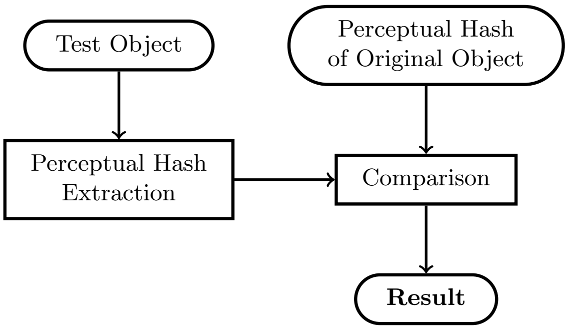

Test Object
Perceptual Hash of Original Object
Perceptual Hash Extraction
Comparison
Result

Figure 2.4: Common architecture of the "integrity verification" usage mode. Cp. [6].

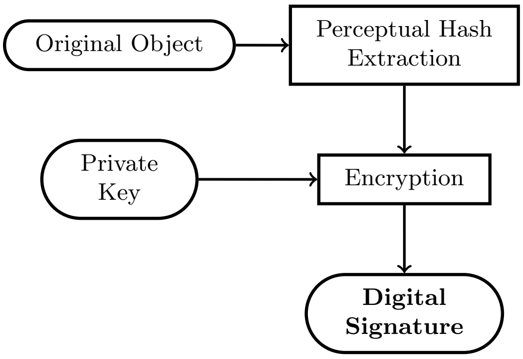


Figure 2.5: Creation of a digital signature.

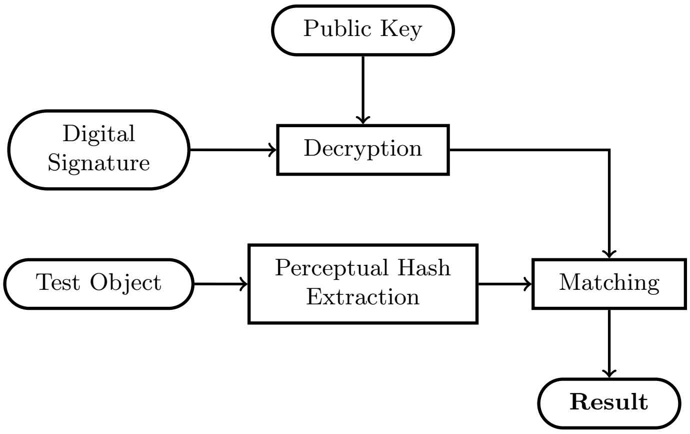


Figure 2.6: Verification of a digital signature.

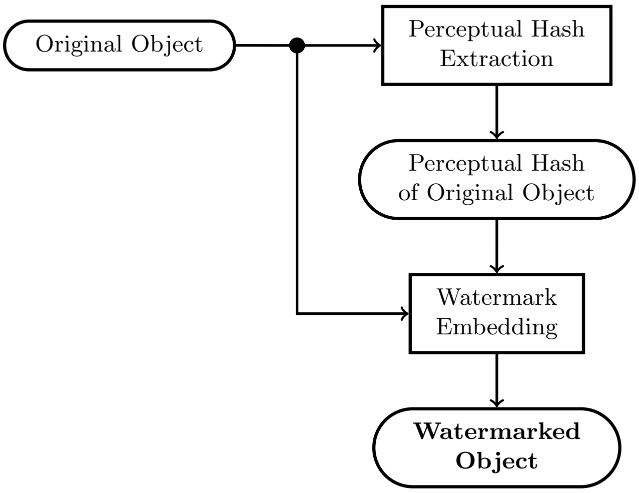

Original Object
Perceptual Hash Extraction
Perceptual Hash of Original Object
Watermark Embedding
Watermarked Object

Figure 2.7: "Self-embedding" integrity verification framework: Embedding. Source: [11].

Watermarking Support

Perceptual hash functions can be used to construct semi-fragile signatures. A perceptual hash is embedded into the media object using a robust or semi-fragile watermark[4] (see Figures 2.7 and 2.8). Semi-fragile watermarks can only authenticate image features they are embedded within. For example, semi-fragile watermarks for images can be implemented by embedding their information in the high-frequency coefficients of the block discrete Cosine transform (DCT). Embedding them in the low-frequency coefficients is not an option because changes in these coefficients can be perceived relatively easily by the human eye. This means that only changes in the high frequency coefficients can be detected by the watermark. These limitations can be overcome by using semi-fragile signatures. A suitable robust or semi-fragile watermarking algorithm can embed any desired perceptual hash. Thus, any image feature considered by the perceptual hashing algorithm can be authenticated. If a semi-fragile watermarking algorithm is used, it can complement the perceptual hash. [15] proposes such semi-fragile signatures. Another advantage when combining perceptual hashes with digital watermarks to achieve authentication is that no database or special dedicated file headers are needed.

According to [30], perceptual hash functions can complement digital watermarks in various other ways:

> *"Audio Fingerprinting can assist watermarking. Audio Fingerprints can be used to derive secret keys from the actual content.*

[4] Robust and semi-fragile watermarks are discussed in section 2.3.1.

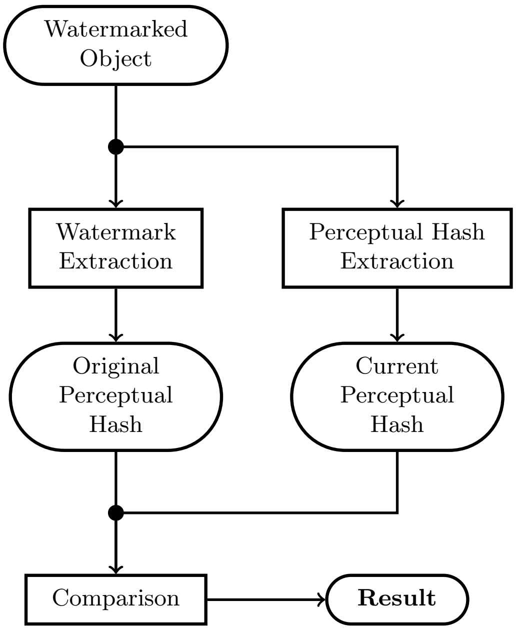


Figure 2.8: "Self-embedding" integrity verification framework: Comparison. Source: [11].

> *As described by Mihçak and Venkatesan (2001) [29], using the same secret key for a number of different audio items may compromise security, since each item may leak partial information about the key. Audio fingerprinting / perceptual hashing can help generate input-dependent keys for each piece of audio. Haitsma and Kalker (2002b) [17] suggest audio Fingerprinting to enhance the security of watermarks in the context of copy attacks. Copy attacks estimate a watermark from watermarked content and transplant it to unmarked content. Binding the watermark to the content can help to defeat this type of attacks. In addition, Fingerprinting can be useful against insertion/deletion attacks that cause desynchronization of the watermark detection: by using the Fingerprint, the detector is able to find anchor points in the audio stream and thus to resynchronize at these locations (Mihçak and Venkatesan, 2001 [29])."*

[39] presents another approach to audio watermarking synchronization. A perceptual audio hash is used to identify watermarking positions. A watermark can be attacked by moving the embedded information to a position where the watermark detection algorithm will not try to retrieve it. Hence the embedded watermark information is not removed from a media object but only displaced slightly. This attack is known as the de-synchronization

attack. This attack can be implemented by time strechting the audio signal. As outlined in [39] that is, *"[...] the slight increase or decrease of audio playing time without pitch modification or significant quality loss [...]."* To increase the robustness against such audio de-synchronization attacks, one solution is to implement repetitive re-synchronizations. Unfortunately re-synchronization in audio watermarking usually requires much of the capacity of the watermark. Therefore, frequent re-synchronization renders a watermark algorithm more robust but also useless due to only minimal capacity. The algorithm proposed by [39] *"[...] does not require embedded sync sequences to synchronize the watermarking bits but uses robust audio hashing technology to re-sync at each embedded bit."*

Content-Based Media Retrieval and Processing

Content-based media retrieval is a generic term for other fields of research like Content-based Image Retrieval (CBIR). [30] highlights that perceptual hash functions can be used for content-based media retrieval and processing:

> *"Deriving compact signatures from complex multimedia objects is an essential step in Multimedia Information Retrieval. Fingerprinting can extract information from the audio signal at different abstraction levels, from low level descriptors to higher level descriptors. Especially, higher level abstractions for modelling audio hold the possibility to extend the Fingerprinting usage modes to content-based navigation, search by similarity, content-based processing and other applications of Music Information Retrieval. In a query-by-example scheme, the Fingerprint of a song can be used to retrieve not only the original version but also "similar" ones (Cano et al., 2002b) [7]."*

2.1.2 Distance/Similarity Functions for Perceptual Hashes

A perceptual hash function calculates similar perceptual hash values for similar media objects. To compare two perceptual hashes appropriate measures must be used. The most often used are the Bit Error Rate (BER), the Hamming distance and the Peak of Cross Correlation (PCC). The first two measure the distance between two hash values, whereas the latter measures the similarity between two hash values. The next sections discuss these measures.

Bit Error Rate (BER)

Definition 2.2 (Bit Error Rate (BER)):
[44] defines the BER ρ as the number i of bit errors of the perceptual hash normalized by the length k of the perceptual hash:

$$\rho := \frac{i}{k},$$

whereas $i \in \{0, 1, \ldots, k\}$ *and* $0 \leq \rho \leq 1$*.*

The number of the bit errors i equals the hamming distance of the perceptual hash values. When comparing perceptually different images the BER should be approximately 0.5. At least, this is the BER that can be expected when comparing two perceptual hash values drawn from a uniform random distribution of $\{0, 1\}^n$. Perceptually equal images should yield a BER close to 0.

Hamming Distance

The hamming distance, as defined in [19, p. 154], is a measurement for the difference of two strings. Such strings can be e.g. binary coded numbers, but they might as well consist of elements from other number systems or alphabets (see table 2.1 for some examples).

String 1	String 2	Hamming distance
00101	**1**0101	1
1**2**34**5**	1**3**34**4**	2
w**e**ll	w**a**ll	4

Table 2.1: Examples of calculating the hamming distance. The strings are from three different alphabets (binary system, decade system and latin alphabet).

Definition 2.3 (Hamming distance):
Let A *denote an alphabet of finite length.* $x = (x_1, \ldots, x_n)$ *denotes an even-length string, whereas* $x \in A$*. The same holds true for* $y = (y_1, \ldots, y_n)$*. Then the hamming distance* Δ *between* x *and* y *is defined as*

$$\Delta(x, y) := \sum_{x_i \neq y_i} 1, i = 1, \ldots, n. \tag{2.5}$$

Definition 2.4 (Normalized hamming distance):
To facilitate comparison, the hamming distance can be normalized with respect to the length n *of the strings. [40] defines the normalized hamming distance* Δ_n *as*

$$\Delta_n(x, y) := \frac{1}{n} \sum_{x_i \neq y_i} 1, i = 1, \ldots, n. \tag{2.6}$$

To calculate the hamming distance of binary coded numbers a XOR operation can be used. Let a and b denote two binary coded numbers of equal length. Then the hamming distance is equal to the number of ones in $a \oplus b$.

Definition 2.5 (Equality Percentage (EP)):
Another metric, as defined in [42], that can be derived is the Equality Percentage (EP):

$$EP := 100 \cdot \Delta_n. \tag{2.7}$$

For perceptually similar images, EP should be high ($\approx 100\%$). Conversely, for perceptually distinct images EP should be low ($\approx 0\%$). Again, the expected value of the EP for two perceptual hash values drawn from a uniform random distribution of $\{0, 1\}^n$ is approximately 50%.

Peak of Cross Correlation

Definition 2.6 (Correlation):
The correlation between two signals is defined as

$$r_{xy}(T) = \int_{-\infty}^{\infty} x(t)y(t+T)dt, \qquad ([42], 2.7)$$

where $x(t)$ and $y(t)$ are two deterministic, real functions[5]*. The correlation function $r_{xy}(T)$ describes the concurrence of these two signals with respect to the offset time T.*

The value of T determines by how much the second signal is shifted to the left. If a signal is correlated with itself, the corresponding function is called **auto correlation function**. If both signals are different, the corresponding function is called **cross correlation function**.

Definition 2.7 (Normalized cross-correlation):
If you have two series x_i and y_i, where $i = 0, 1, 2, \ldots, N-1$ and N denotes the length of both series, then the normalized cross-correlation r at delay d is defined as (cmp. [3] and [23]):

$$r_d = \frac{\sum_i[(x_i - mx) \cdot (y_{i-d} - my)]}{\sqrt{\sum_i (x_i - mx)^2} \cdot \sqrt{(y_{i-d} - my)^2}}, \tag{2.8}$$

where mx and my are the means of the corresponding series.

[5] This means that $x(t)$ and $y(t)$ can take any (real) values. By contrast, a digital filter can assume only a finite number of possible amplitude values.

The PCC is the maximum correlation that can be achieved between these two series.

2.2 Cryptographic Hash Functions

Depending on the application for which a cryptographic hash function is used, it has to meet certain requirements. A cryptographic hash function that will be used as a Modification Detection Code (MDC) in an asymmetric signature application (e.g. a RSA signature facilitating SHA-1) must exhibit the following properties (where H denotes an unkeyed hash function with inputs x, y and outputs x', y')[27, p. 327]:

1. preimage resistance (also named "one-way") – for essentially all prespecified outputs, it is computationally unfeasible to find *any input* which hashes to *that output*, i.e., to find any preimage x such that $h(x) = x'$ when given any x' for which a corresponding input is not known.
2. 2nd-preimage resistance (also named "weak collision resistance") – it is computationally unfeasible to find *any second input* which has the same output as *any specified input*, i.e., given x, to find a 2nd-preimage $y \neq x$ such that $h(x) = h(y)$.
3. collision resistance (also named "strong collision resistance") – it is computationally unfeasible to find *any two distinct inputs* x, y (both inputs can be chosen freely) which hash to the same output, i.e., such that $h(x) = h(y)$.

2.2.1 Application Scenarios

Cryptographic hash functions have many applications nowadays. The required properties cause to change the output dramatically even if only one bit of the input changes. Therefore these hash algorithms are ideally suited to verify the integrity of binary data. Another application is the storage of passwords. Normally operating systems or programs do not store user passwords in clear-text but instead a hash of the users' passwords. Most digital signature schemes also make heavy use of cryptographic hash functions. Actually they do not sign the messages itself but only their hash values.

2.3 Digital Watermarks

Contrary to perceptual hash functions, watermarking algorithms embed information directly into the content (see Figure 2.9). Therefore, the watermarking algorithm has to modify the content. As a consequence, only

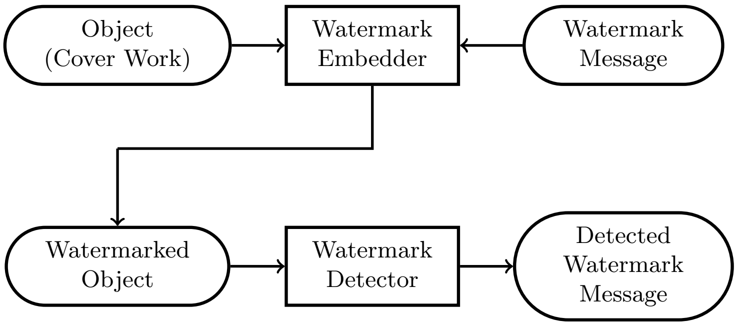


Figure 2.9: A generic watermarking system. Cp. [11].

content that has been watermarked beforehand can be identified or checked for its authenticity. In contrast to this, perceptual hash functions can also identify or authenticate content that has been previously distributed without any attached labels (embedded watermarks). This highlights an advantage of perceptual hash functions. If a watermarking algorithm is compromised (e.g. an adversary is able to remove the embedded watermark from an image and distributes the image) the content can no longer be identified or authenticated. If a perceptual hash function is compromised, the copyright holder can switch to another function. After the database of hashes has been updated using the new function, the copyright holder can continue to e.g. identify his content.

Nevertheless, the insertion of information into the multimedia content offers also benefits over perceptual hashing. Watermarking algorithms can embed additional information (e.g. name of the copyright holder, serial number, identification number of the customer who bought the content) into the multimedia content. Section 2.3.1 illustrates that digital watermarking and perceptual hashing can also be combined.

Watermarking algorithms can be divided into two categories.[11, sec.1.1] Perceptible (non-steganographic) watermarks do not keep the embedded information secret whereas imperceptible (steganographic) watermarks do. Perceptible watermarks can be used e.g. by a photographer who wants to provide a noticeable piece of evidence that he is the copyright holder of an image. Imperceptible watermarks are embedded in such a way that the quality of the content is not modified in a noticeable way.

2.3.1 Application Scenarios

[11, sec. 2.1] identifies various application scenarios. In this section a brief discussion of the most important ones follows.

Owner Identification

Contrary to e.g. a textual copyright notice, which can be forged easily, an appropriate digital watermark can prove ownership. Only the legitimate copyright holder can detect and eventually remove a watermark.

Broadcast Monitoring

The goal of broadcast monitoring is to supervise and/or backtrack multimedia content that is broadcast via e.g. radio or television. One actual application for advertisers is to verify automatically if a commercial was broadcast as contracted. The necessity for such controls was highlighted in 1997. Some Japanese television broadcasters routinely overbooked their air time. Advertisers paid for commercials that were never aired.[20] The fraud was discovered through manual broadcast monitoring[6]. Automated broadcast monitoring can be divided into two categories.[11, sec. 2.1.1] Broadcast monitoring systems that rely on additional information that is broadcast along with the actual multimedia content are called **active monitoring systems**. Systems trying to identify the content itself, without the help of additional information, are named **passive monitoring systems**. Therefore, broadcast monitoring systems utilizing watermarking techniques are active monitoring systems, whereas systems using perceptual hashing are passive monitoring systems.

Transaction Tracking

Watermarks can be used to identify customers who illegally leaked multimedia content to the press or uploaded it to internet file sharing platforms such as the eDonkey network[7].

Integrity Verification

The increasing performance of personal computers and the availability of more and more sophisticated applications has made the manipulation of digital multimedia content incredibly easy to perform and increasingly difficult to detect.[16] To verify the integrity of multimedia content, digital watermarks can be utilized.

When used for integrity verification, digital watermarks can be combined with perceptual hash functions. Hence this topic will be discussed in greater depth in this section. [11, sec. 10.5] distinguishes the following watermarks that can be used for integrity verification:

[6] Human observers watched the television programs of the broadcast stations and controlled if and when their commercials where actually aired.

[7] The eDonkey network is discussed in [31].

1. Fragile watermarks become undetectable if the slightest modification or manipulation is applied to their carrier.
2. Embedded signatures are cryptographic signatures embedded as watermarks.
3. Semi-fragile watermarks are designed to survive legitimate distortions (modifications) but to be destroyed by illegitimate ones (manipulations).
4. Semi-fragile signatures are based on perceptual hash functions and are discussed in section 2.1.1.
5. Tell-tale watermarks can be examined after the carrier has been modified or manipulated to discover in which way the carrier was changed.

Exact integrity verification systems aspire to verify that the carrier (also known as "cover work") of a watermark has not been tampered with at all. Such systems can make use of the techniques 1 – 2. Selective integrity verification systems aspire to verify that the carrier of a watermark has not been manipulated by any of a predefined set of illegitimate distortions, while allowing modification by legitimate distortions. Such systems can make use of the techniques 3 – 5.

2.4 Relationship of Discussed Techniques

The examples of application scenarios illustrate that the discussed techniques (perceptual hashing, cryptographic hashing and digital watermarking) are not mutually exclusive. For certain applications there is no clear answer as to which of these techniques to use. For broadcast monitoring, perceptual hashing or digital watermarks can be employed, for instance. Staying with this example, it depends on the actual application scenario to decide which of these techniques to use. To acquire market research data (e.g. to estimate how much air time a company buys at local television broadcast stations), perceptual hashing is better suited. Anyway, to employ perceptual hashing for broadcast monitoring, no help or consent from the broadcasters or advertisers is needed at all. Conversely, to verify that a television broadcast station airs all the commercials an advertising company has bought, digital watermarks are more appropriate. The reason is because passive monitoring systems are less accurate than active ones.[11, sec. 2.1.1] [10] discusses the relationship between perceptual hash functions, cryptographic hashing algorithms and digital watermarking in greater depth. [8] evaluates the security of some perceptual hash functions.

Chapter 3

Perceptual Image Hash Functions

3.1 Theoretical Discussion

3.1.1 DCT Based Hash

The DCT, like any Fourier-related transform, expresses a function or signal (a sequence of finitely many data points) in terms of a sum of sinusoids with different frequencies and amplitudes. The DCT uses only cosine functions, while e.g. the discrete Fourier transform (DFT) uses both cosines and sines. There are eight different standard variations of the DCT. The most common variant is the type-II DCT. Therefore it is often simply referred to as DCT.

Definition 3.1 (Type-II DCT):
Let $x[m]$, $m = 0, \ldots, N-1$, denote an N-point real signal sequence. Then [13] defines the type-II DCT as

$$X[n] = \sqrt{\frac{2}{N}} \cdot \sum_{m=0}^{N-1} x[m] \cdot cos\Big(\frac{(2m+1) \cdot n\pi}{2N}\Big) \\ ,(n = 0, \ldots, N-1). \tag{3.1}$$

This can also be expressed as

$$X[n] = \sum_{m=0}^{N-1} c[n,m] \cdot x[m] \\ ,(n = 0, \ldots, N-1), \tag{3.2}$$

where $c[n,m]$ denotes the row number n and column number m of the DCT matrix.

Definition 3.2 (DCT matrix):
The DCT matrix is defined as

$$c[n,m] = \sqrt{\frac{2}{N}} \cdot cos\Big(\frac{(2m+1)\cdot n\pi}{2N}\Big) \\ ,(m,n = 0,\dots,N-1). \tag{3.3}$$

Equation 3.2 is especially useful when the DCT has to be implemented programmatically. The DCT matrix (equation 3.3) can be calculated in advance for any given size N.

The DCT is a separable linear transformation. The two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension. So if image I is square, the two-dimensional DCT of I can be computed as (M denotes the DCT matrix)

$$\text{DCT}(I) = M \cdot I \cdot M'. \tag{3.4}$$

Various properties of the DCT can be utilized to create perceptual image hash functions. Low-frequency DCT coefficients of an image are mostly stable under image manipulations.[14] That is because most of the signal information tends to be concentrated in a few low-frequency components of the DCT. This property is also utilized by the JPEG image compression standard.[4][ch. 7] There, the two-dimensional type-II DCTs of NxN pixel blocks are computed and the results are quantized. N is typically 8 and the type-II DCT formula is applied to each row and column of the block. The result is an $8x8$ transform coefficient array in which the elements close to the top-left (index position $(0,0)$) represent low-frequency components and are therefore deemed to be perceptually most significant. Coefficients with increasing vertical and horizontal index values represent higher vertical and horizontal frequency components. [24] shows that a feature code can be extracted from the relationship between two DCT coefficients of the same position in two separate blocks. This property is especially useful for image integrity verification systems, which are expected to pass only JPEG compression. In summary, that is because all DCT coefficient matrices are divided by the same quantization table in the JPEG compression process.

3.1.2 Marr-Hildreth Operator Based Hash

Several perceptual image hash functions that use edge detectors for feature extraction have been proposed (e.g. [1]). To facilitate the discussion of such algorithms gradient and Laplacian based edge detection are discussed first. [4] outlines that,

> *"although the precise defnition depends on the application context, an edge can generally be defined as a boundary or contour that separates adjacent image regions having relatively distinct characteristics according to some feature of interest."*

These features of interest can be colour or texture, but most commonly grey level (or luminance) is used. The result of an edge detection process is typically an **edge map**. An edge map describes each original pixel's edge classification and perhaps additional edge attributes, like magnitude and orientation. If an edge is defined as an abrupt grey level change then the derivative, or **gradient**, can be used for edge detection. Suppose $f_c(x)$ denotes the grey level function of a line (a one-dimensional array of pixels). An edge therefore can be seen as the transition from a low to a high amplitude or vice versa. The **gradient approach** to edge detection, therefore, is to locate the positions where the first derivative of $f_c(x)$ reaches a local extremum. Another approach for edge detection is to use the second derivative of $f_c(x)$. The **Laplacian approach** is to locate the positions where zero-crossings of $f_c''(x)$ occur. These two approaches can be adapted for discrete, two dimensional images, but certain adjustments have to be made. First, edges in two dimensional images have the additional property of direction. For some applications, a directionally-sensitive edge detector is useful. Additionally, the discrete nature of digital images requires the use of an approximation to the derivative. Finally there are a number of problems that can impair the edge detection process in "real" images. The most prominent one is noise. The derivative operator acts as a highpass filter. Consequently, edge detectors based on it are sensitive to noise. The wide variety of edge detection algorithms that have been developed exist mostly because of the many different ways proposed for dealing with noise and its effect. Also there is a trade-off between the correct detection of actual edges and the detection of their precise location. Detection errors, as previously mentioned, tend to increase with noise. Therefore, noise suppression is very important in achieving a high detection accuracy. [4] outlines that *"In general, the potential for noise suppression improves with the spatial extent of the edge detection filter."* Consequently, to achieve a high detection accuracy, a large-sized filter is preferable. Conversely, to achieve good localization, the filter should be of small spatial extent. The rest of this section focuses on the discussion of Laplacian-based methods.

Definition 3.3 (Continuous Laplacian):
Let $f_c(x,y)$ denote the grey level function of an image. Then the continuous Laplacian is defined as

$$\nabla^2 f_c(x,y) = \nabla \cdot \nabla f_c(x,y) = \frac{\partial^2 f_c(x,y)}{\partial x^2} + \frac{\partial^2 f_c(x,y)}{\partial y^2}. \qquad ([4], 3.4)$$

The zero-crossings of $\nabla^2 f_c(x, y)$ occur at the edge points of $f_c(x, y)$ because of the second derivative. Laplacian-based edge detection produces edges of zero thickness. Edge-thinning steps, like those required by Gradient-based methods, are therefore not neccessary. Different filters (discrete Laplacian operators) can be constructed from the continuous Laplacian. Such a filter, $h(n_1, n_2)$, can be applied to a discrete-space image by using convolution. The Laplacian estimate for an image, $f(n_1, n_2)$, is then

$$\hat{\nabla}^2 f(n_1, n_2) = f(n_1, n_2) * h(n_1, n_2), \qquad ([4], 3.4)$$

where $*$ denotes convolution. To actually get an edge map, another processing step is neccessary. The zero-crossings in the discrete-space image $\nabla^2 f(n_1, n_2)$ have to be located. [4] suggests that each image pixel should be compared to its eight neighbours. If a pixel p differs in sign with its neighbor q, then an edge lies between them. That is, pixel p is classified as a zero crossing if

$$|\nabla^2 f(p) \leq \nabla^2 f(q)|. \qquad (3.5)$$

The Marr-Hildreth operator, also denoted as the Laplacian of Gaussian (LoG), is a special case of a discrete Laplace filter. The filter kernel is constructed by applying the Laplace operator onto a Gauss function. Because of its form, it is also called “mexican hat” filter. That is, because when visualized in three dimensions it looks like a sombrero hat. The LoG can be tuned to detect edges at a particular scale. In [4] the importance of this property is outlined as follows:

> *“It is common for a single image to contain edges having widely different sharpnesses and scales, from blurry and gradual to crisp and abrupt. Edge scale information is often useful as an aid toward image understanding. For instance, edges at low resolution tend to indicate gross shapes while texture tends to become important at higher resolutions. An edge detected over a wide range of scale is more likely to be physically signifcant in the scene than an edge found only within a narrow range of scale. Furthermore, the effects of noise are usually most deleterious at the finer scales.”*

Definition 3.4 (Gaussian filter):
Omitting the scaling factor the Gaussian filter is defined in [4] as

$$g_c(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}}. \qquad (3.6)$$

The convolution and the Laplacian operations can be interchanged:

$$\nabla^2[f_c(x,y) * g_c(x,y)] = [\nabla^2 g_c(x,y)] * f_c(x,y). \qquad \text{([4, p. 513], 3.6)}$$

The derivative and the convolution are both linear operators. Consequently, Gaussian filtering ($g_c(x,y)$) followed by differentiation is the same as filtering with the derivative of a Gaussian ($[\nabla^2 g_c(x,y)]$). This allows an computational efficient implementation. $\nabla^2 g_c(x,y)$ can be prepared in advance, because it does not depend on the image ($f_c(x,y)$).

Definition 3.5 (Laplacian of Gaussian (LoG) filter):
The Laplacian of Gaussian (LoG) filter, denoted as $h_c(x,y)$, can be defined as

$$\begin{aligned} h_c(x,y) &= \nabla^2 g_c(x,y) \\ &= \frac{x^2+y^2-2\sigma^2}{\sigma^4} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}. \end{aligned} \qquad \text{([4, p. 513], 3.6)}$$

To implement the LoG in discrete form, one may construct a filter by sampling equation [4, p. 513], 3.6 after selecting an value for σ. The filter then may be applied to an image by using 2D convolution. The computational complexity can be further decreased by using 1D convolution. That is possible because the discrete form of equation [4, p. 513], 3.6 is actually the sum of two separable filters. The Gaussian functions itself is a separable function. Therefore, as outlined in [4],

> *"by constructing and applying the appropriate 1D filters successively to the rows and columns of the image, the computational expense of 2D convolution becomes unnecessary. Separable convolution to implement the LoG is roughly 1–2 orders of magnitude more efficient than 2D convolution. If an image is $M \times M$ in size, the number of operations at each pixel is M^2 for 2D convolution and only $2M$ if done in a separable, 1D manner."*

Furthermore [4] proposes to work in the frequency domain instead of the spatial domain. This approach is more efficient if the filter extent is not small.

The LoG (equation [4, p. 513], 3.6) can also be approximated by the difference of two 2D Gauss functions having properly-chosen scales. The Difference of Gaussian (DOG) filter is

$$h_c(x,y) = g_{c1}(x,y) - g_{c2}(x,y), \qquad \text{([4, p. 515], 3.6)}$$

where $\frac{\sigma_2}{\sigma_1} \approx 1,6$ and g_{c1} and g_{c1} are evaluated using equation 3.6. However, as outlined in [4], *"[...] the LoG is usually preferred because it is theoretically optimal and its separability allows for efficient computation. For the same accurracy of results, the DOG requires a slightly larger filter size."*

3.1.3 Radial Variance Based Hash

A perceptual image hash function based on the Radon transform[34] was proposed by Lefèbvre and Macq in [22] in September 2002. A few years later, in April 2005, both authors outlined in [38] that their previously proposed algorithm suffers from some troubles. Thereupon they introduced a new algorithm (see [38] and [35]) to overcome these problems.

The Radon transform is the integral transform consisting of the integral of a function over a straight line. It is robust against various image processing steps (e.g. compression) and geometrical transformations (e.g. rotation). In [34] a new visual content descriptor, based on the Radon transform, was presented. Let α denote the angle of the used projection line. x denotes the coordinate of a pixel along the x-axis, whereas y denotes the coordinate of a pixel along the y-axis. To extend the Radon transform to discrete images, the line integral along $d = x \cdot cos\alpha + y \cdot sin\alpha$ can be approximated by a summation of the pixels lying in the one pixel wide strip:

$$d - \frac{1}{2} \leq x \cdot cos\alpha + y \cdot sin\alpha \leq d + \frac{1}{2}. \qquad ([38], 3.6)$$

The algorithm proposed in [38] uses the variance instead of the sum of the pixel values along the line projections. The variance captures luminace discontinuities along the projection lines much better. Such discontinuities result from edges, that are orthogonal to the projection direction. The so-called radial variance vector ($R[\alpha]$) is therefore defined as follows. Let $\Gamma(\alpha)$ denote the set of pixels (x, y) on the projection line corresponding to a given angle α. Let (x', y') denote the coordinates of the central pixel of the image. $(x, y) \in \Gamma(\alpha)$ if

$$-\frac{1}{2} \leq (x - x') \cdot cos\alpha + (y - y') \cdot sin\alpha \leq \frac{1}{2}. \qquad (3.7)$$

Definition 3.6 (Radial variance vector):
Let $I(x, y)$ denote the luminance value of the pixel (x, y), the radial variance vector $R[\alpha]$, where $\alpha = 0, 1, \ldots, 179$, is then defined by

$$R[\alpha] = \frac{\sum_{(x,y)\in\Gamma(\alpha)} I^2(x, y)}{\#\Gamma(\alpha)} - \Big(\frac{\sum_{(x,y)\in\Gamma(\alpha)} I(x, y)}{\#\Gamma(\alpha)}\Big)^2. \qquad (3.8)$$

As discussed in [22], it is sufficient to extract 180 instead of 360 values. That is because the Radon transform is symmetric. Finally, in [35], the perceptual image hash function was further improved by applying the DCT to the radial variance vector. The first 40 coefficients of the transformed radial variance vector form the so-called radial hash vector in the end. This omits redundant components of the radial variance vector and efficiently decorrelates it.

3.1.4 Block Mean Value Based Hash

In 2006, Bian Yang, Fan Gu and Xiamu Niu proposed a block mean value based perceptual image hash function in [44]. Four slightly different methods are proposed. The latter two additionally incorporate an image rotation operation to enhance robustness against rotation attacks. This significantly increases the computational complexity of the latter two methods. To secure the perceptual image hash values encryption using a secret key is used.

Method 1

The first method is described as follows:

a) Convert the image to grey scale and normalize the original image into a preset size.

b) Let N denote the bit length (e.g. 256 bit) of the final hash value. Divide the pixels of the image I into non-overlapped blocks $I_1, I_2, \ldots, I_N$.

c) Encrypt the indices of the block sequence $\{I_1, I_2, \ldots, I_N\}$ using a secret key K to obtain a block sequence with a new scanning order $\{I'_1, I'_2, \ldots, I'_N\}$. [44] specifies no further details about what encryption algorithm to use. So it is up to the implementor of this perceptual image hash function to choose an adequate one.

d) Calculate the mean of the pixel values of each block. That is, calculate the mean value sequence $\{M_1, M_2, \ldots, M_N\}$ from corresponding block sequence $\{I'_1, I'_2, \ldots, I'_N\}$. Finally obtain the median value M_d of the mean value sequence.

e) Normalize the mean value sequence into a binary form and obtain the hash value h as

$$h(i) = \begin{cases} 0 & , M_i < M_d \\ 1 & , M_i \geq M_d \end{cases}. \tag{3.9}$$

Method 2

The only difference to the first method is that the pixels of the image are divided into **overlapped** blocks. The degree of overlapping is set to be half the size of a block. If a preset size of 16x16 pixels is chosen and a block size of 4x4 pixels is used, the first method would yield a hash with a bit size of 16 bits. Using this method the pixels of the image would be divided into 49 blocks. Therefore the hash would have a size of 49 bits.

Method 3

The third method offers more robustness against rotation attacks. The difference is that the mean values of the pixel blocks are rotated several times:

a) Perform steps a – d from the first method.

b) Rotate by D degrees the matrix M formed by $\{M_1, M_2, \ldots, M_N\}$, whereas $D = \{0, 15, 30, \ldots, 345\}$. This yields 24 matrices (M_i, ($i = 1, 2, \ldots, 24$)). Divide each of the 24 rotated matrices into N blocks. Obtain the mean value sequence $\{M_{i1}, M_{i2}, \ldots, M_{iN}\}$ of each block and median value M_{di} of this sequence, which forms 24 groups of sequences.

   [44] does not outline which matrix rotation operation to use. The author of this thesis suggests using a plain image rotation operation using no interpolation. Furthermore, the matrices (in fact the images) must not be enlarged by the rotation operation because all the matrices are required to have the same dimensions.

c) Perform equation 3.9 for the 24 groups of sequences and obtain the final hash value matrix.

Method 4

The fourth method is a combination of the second and third method. The image is divided into overlapping blocks like described in the second method. Furthermore, the hash is calculated using the rotated mean values of the blocks like described in the third method.

3.2 pHash – Discussion of an Implementation

The previous section discussed and reviewed the theoretical background behind some perceptual image hash functions. This section discusses actual implementations of these functions. There are hardly any implementations of perceptual image hash functions publicly available. The Image Hashing Toolbox is a set of modules written for Matlab which implement the following perceptual image hash functions:

- Discrete wavelet transform (DWT) based hash
- Hashing via singular value decomposition (SVD)
- Hashing using feature points

Another implementation of perceptual image hash functions is available in the form of a C/C++ library. The library is called pHash. It implements all different sorts of perceptual hash functions. Textual or audio hash functions are also provided for instance. pHash implements the following perceptual image hash functions:

- DCT based hash
- Radial variance based hash
- Marr-Hildreth operator based hash

pHash also offers functions to store, query and retrieve perceptual hash values in a performant way. The following sections discuss the parts of the pHash Application Programming Interface (API) a programmer can use to create and compare perceptual image hashes. Section 3.2.4 discusses a perceptual image hash function that was implemented in pHash as part of this thesis. Listing 3.1 shows how to compile the pHash library. Listing B.1 depicts the functions a programmer has to use to create and compare different perceptual image hash values.

```
pHash $ ./configure --enable-debug --enable-java
pHash $ export CXXFLAGS="-O0 -ggdb"
pHash $ export CFLAGS="-O0 -ggdb"
pHash $ make
```

Listing 3.1: Compilation of pHash in debug mode under GNU/Linux.

3.2.1 DCT Based Hash

The API function to use is `ph_dct_imagehash()`. This function calculates a fixed length (64 bit / 8 Byte) hash. The hash is “returned” in the variable `hash`. `file` is a string variable containing the name of the file to hash. The return value of the function is an integer, where 0 indicates success and -1 indicates a failure.

The actual calculation of the hash value takes place in the function `ph_dct_imagehash()` itself. The hash is stored in an unsigned 64 bit integer. It is a binary sequence. To measure the distance between two hash values the hamming distance is used. Function `ph_hamming_distance()` implements the calculation of the hamming distance for this type of hash.

The pHash implementation is actually inspired by a DCT based perceptual video hash function. The video hash function was published in [9]. The method `ph_dct_imagehash()` first converts the image to grey scale using only its luminance. This step is common to all perceptual image hash functions, because the essential semantic information resides in the luminance component of an image. Then a mean filter[1] is applied to the image. A kernel with dimension $7x7$ is used. To apply this kernel, the `get_convolve()` function of the CImg library (see equation 3.12) is used. After this operation the image is resized to $32x32$ pixels. Consequently, a DCT matrix is generated and the two-dimensional type-II DCT coefficients are calculated using matrix multiplications. The image is square. Therefore the two-dimensional DCT can be computed by multiplying the DCT matrix with the image and the transposed DCT matrix.

As proposed in [9], 64 low-frequency DCT coefficients, omitting the lowest frequency coefficients, are selected for hash extraction. pHash therefore selects $8x8$ transform coefficients. The selected coefficients form a square matrix. The coefficient $\mathrm{DCT}(1,1)$ being the upper left corner of the matrix and the coefficient $\mathrm{DCT}(8/8)$ being the lower right corner of the matrix. The rows of the square matrix are stringed together forming a one-dimensional array of length 64. Let the DCT coefficients of the array be denoted as $C_i, i = 0, \ldots, 63$. Once the median m of the 64 DCT coefficients has been determined, the sequence can be normalized into a binary form as follows to form the final hash value

$$h_i = \begin{cases} 0 & , C_i < m \\ 1 & , C_i \geq m \end{cases}, \tag{3.10}$$

where h_i is the bit of the perceptual image hash at position i.

3.2.2 Marr-Hildreth Operator Based Hash

The method `ph_mh_imagehash()` calculates a fixed length (576 bit / 72 byte) hash. A pointer to the hash value is the return value of this function. Although the length of the hash is fixed, its length is “returned” in the variable `N`. Again, the variable `filename` is a string variable containing the name of the image file to hash. The variable `alpha` is the scale factor for the Marr-Hildreth operator (default is 2). The variable `lvl` is the level of the scale factor (default is 1).

[1] Other common names are smoothing, averaging or box filter.

The actual calculation of the hash value takes place in the function `ph_mh_imagehash()` itself. The LoG kernel is applied to the image using the `get_correlate()` function from the CImg library.

Definition 3.7 (CImg correlation):
Let x, y, z denote the pixel width, height and depth of an image I. Let i, j, k denote the pixel width, height and depth of a mask M. The result R of the correlation of an image I by a mask M is then defined by CImg to be:

$$R(x, y, z) = \sum_{i,j,k} I(x + i, y + j, z + k)M(i, j, k) \tag{3.11}$$

Definition 3.8 (CImg convolution):
pHash also implements a convolution operation such that the result R of the convolution of an image I by a mask M is to be :

$$R(x, y, z) = \sum_{i,j,k} I(x - i, y - j, z - k)M(i, j, k) \tag{3.12}$$

The hash is stored in an `uint8_t` array containing a binary sequence. The normalized hamming distance is used to measure the distance between two hash values. The function `ph_hammingdistance2()` implements the calculation of the normalized hamming distance for such a hash type.

The pHash implementation has not been proposed previously. The authors rather implemented their own approach with reagard to e.g. feature extraction. Before feature extraction, various pre-processing steps are applied to the image. First and foremost, the image is converted to grey scale. Then it is blurred using a Canny-Deriche filter. The sigma of the filter is set to 1.0. After that, the image is resized to a resolution of 512 x 512 pixels. Finally a histogramm-equalized version of the image is calculated using 256 histogram levels.

3.2.3 Radial Variance Based Hash

The method `ph_image_digest()` calculates a fixed length (320 bit / 40 byte) hash. The hash is “returned” in the structure `digest`. See below for more information on this structure. `file` is a string variable containing the path of the image to hash. The `sigma` is the deviation for the gaussian filter. The `gamma` is the value used for gamma correction on the input image. Although there are no default values given for `sigma` and `gamma`, the authors suggest 1 for both variables.[21] `N` is the number of angles to consider (default is 180).

The structure `digest` represents one hash. The hash value is stored in an `uint8_t` array (`coeffs`). Although the length of the hash is fixed (40 bytes), the member `size` contains the size of the hash in bytes. Each DCT coefficient is stored in an `uint8_t` data type. The actual hash calculation takes place in ph_dct(). Comparing two hash values is done by calculating the PCC between the two hash values. This is implemented by function ph_crosscorr(). The PCC is "returned" in the parameter pcc. x and y are the two hashes to compare. The function also determines if the PCC is above or below a given threshold (variable threshold, default value is 0.9). If the PCC is above the threshold, the two images are considered to be the same and 1 is returned. Conversely, 0 is returned if the two images are considered to be different.

pHash implements the algorithm as proposed in [35]. At first, the image is converted to grey scale. After that pHash implements a few additional image pre-processing steps. That is, as suggested by the two function parameters sigma and gamma, blurring and gamma correction. Of the discussed perceptual image hash functions, the radial variance based image hash function is the only one which does not normalize the image with respect to resolution. None of the papers that proposed radial variance based hash functions ([22], [38], [35]) discusses any image normalization operations that may make sense when implementing such a hash function. [22] mentions the term normalization but no further details are outlined.

3.2.4 Block Mean Value Based Hash

As part of this thesis, a block mean value based perceptual image hash function was newly implemented into pHash. The function bmb_imagehash() calculates a variable length hash. The hash is returned in a BinHash object (variable ret_hash). The file is a string variable containing the path of the image. The hashopts is a pointer to a s_bmb_hashopts structure which holds the options to be used by the hash function. method is an integer value used to specify which method the image hash function should use. [44] proposed four slightly different methods of this image hash function. The first two have been implemented. The encryption of the indices of the block sequence using a secret key is omitted by this implementation (step c of method 1 in section 3.1.4).

The actual calculation of the hash value takes place in bmb_imagehash() itself. The BinHash class uses an uint8_t array to store the actual hash value. The hash value is a binary sequence. Therefore the normalized hamming distance is used to measure the distance between two hash values. The function ph_hammingdistance2() implements the calculation of the normalized hamming distance for such a hash type. As outlined in [44] the image is converted to grey scale and resized to a square resolution. The default resolution of the pHash implementation is 256 x 256 pixels.

3.2.5 Java Interface

pHash also provides a Java API. Java does not allow to access classes in the default package from a named package. Because Rihamark resides in its own packages and the pHash Java Native Interface (JNI) implementation resided in the default package, the pHash Java bindings had to be refactored. The resulting patch was also posted to the pHash mailing list.[2] Listing 3.2 shows how to compile the Java classes of pHash, generate C/C++ header files for them and finally compile the required C/C++ pHash libraries. Listing B.2 shows the JNI calls related to ressource management and DCT image hashing and the related API Java programs can use. Java programs can use the `public static` methods of class *pHash* to calculate and compare image hashes.

A caveat when trying to load a JNI library in a Java program is that the Java Virtual Machine does not use the default mechanism of the operating system to locate dynamic libraries. A C/C++ program running on a GNU/Linux based operating system would normally use the dynamic linking loader to load dynamic libraries. To be able to load a dynamic library from within Java, the so-called "Java libary path" must contain the path to the directory of the library. Inside the Java Virtual Machine the Java library path is stored in the `java.library.path` property. The Java library path can only be set using the appropriate command line option when starting the Java Virtual Machine. Under Unix-based operating systems, the content of the `LD_LIBRARY_PATH` environmental variable is merged with the Java library path. Furthermore the Java library path contains the directories */lib/* and */usr/lib/* per default. According to the Filesystem Hierachy Standard[3] the */lib/* directory should contain essential shared libraries and kernel modules. The */usr/lib/* directory should contain libraries for programming and packages. Naturally, a JNI library can reference other dynamically linked libraries. The Java Virtual Machine will then locate the "initial" JNI library using the Java library path, but the "secondary" libraries are loaded using the default mechanism of the operating system.

```
pHash/bindings/java $ javac org/pHash/*.java
pHash/bindings/java $ javah -jni -classpath . org.pHash.pHash
pHash/bindings/java $ javah -jni -classpath . org.pHash.MVPTree
pHash/bindings/java $ make
```

Listing 3.2: Compilation of the pHash Java package and the required JNI bindings under GNU/Linux.

[2] Web front-end for the mailing list archive: `http://lists.phash.org/pipermail/phash-support-phash.org/2010-April/000052.html`, copy on CD-ROM (lit-001).

[3] Homepage: `http://www.pathname.com/fhs/`

Chapter 4

Benchmarking

The performance of perceptual hash functions can be compared using various error rates or error percentages. The following sections review the computation and interpretation of such error types. No work has been published yet that thoroughly discusses benchmarking and error types in the field of perceptual hashing. Because perceptual hash functions are similar to biometric authentication systems – both are just a kind of pattern recognition application – it is feasible to refer to work published in the field of biometric authentication systems when discussing error types for perceptual hash functions. The discussion in this section is mainly based on [2, sec. 5 and sec. 6] and [43, sec 9.2]. It has to be stressed that most of these error types have to be calculated differently depending on whether a perceptual hash function is used for content identification or integrity verification. When used for integrity verification a perceptual hash function makes a one-to-one $(1 : 1)$ match based on a similarity score s. When used for content identification it has to make a one to many $(1 : m)$ match.

4.1 Metrics for Verification Systems

The integrity verification of media objects can be considered as a two-class prediction problem (binary classification), in which the outcomes are labelled either as positive or negative. There are four possible outcomes. If the outcome from a prediction is "authentic" and the actual value is also "authentic", then it is called a **true positive**. But if the actual value is "not authentic" then it is said to be a **false positive**. Conversely, a **true negative** has occurred when both the prediction outcome and the actual value are "not authentic", and a **false negative** has occured when the prediction outcome is "not authentic" while the actual value is "authentic". The confusion matrix in figure 4.1 illustrates the possible outcomes.

The quality of perceptual hash functions can be evaluated on the basis of the number of falsely classified media objects. The False Accept Rate

(FAR) and the False Reject Rate (FRR) are common metrics to specify the probability of falsely classified media objects. They depend on the chosen threshold. The threshold, FAR, FRR and other important metrics are discussed below.

Decision / Attempt	**Authentic (class 1)**	**Not authentic (class 2)**
Accept	True positive	False positive (Type 2 error)
Reject	False negative (Type 1 error)	True negative

Table 4.1: Confusion matrix.

4.1.1 Threshold

When a perceptual hash function compares two media objects, the outcome is a similarity score s. “Similarity score” is a generic term and the actual represenation of such similarity score heavily depends on the perceptual hash function. Many perceptual image hash functions use e.g. the BER as a similarity score when calculating the “distance” between two hash values. If s is smaller than the chosen threshold T then the media objects are predicted to be perceptually similar (everything left of T in figure 4.1). If s is bigger than T then the media objects are predicted to be perceptually different (everything right of T in figure 4.1). Therefore, depending on the threshold, the result set is divided into authentic and not authentic media objects. Consequently, the selection of the threshold is crucial for the application of perceptual hash functions. Section 2.1 already discussed why the selection of a threshold is problematic. Authentic and not authentic media objects can not be separated clearly. The boundary between these two sets is fuzzy.

4.1.2 False Accept and False Reject Rate (FAR/FRR)

For a given threshold, the performance of a perceptual hash function can be calculated on the basis of the falsely classified media objects. Falsely classified objects are either perceptually different objects that are recognized as authentic (FAR) or perceptually identical objects which are recognized as not authentic (FRR). The ideal case would be that all media objects were recognized correctly ($d' \gg 0$, see section 4.1.3). But normally there is no such threshold.

A formal definition of the FAR and FRR follows (cmp. [33] and [2]). Let H_0 denote the null hypothesis and H_a the corresponding alternative hypothesis:

H_0: The compared perceptual hashes are from perceptual identical media objects.

H_a: The compared perceptual hashes are **not** from perceptual identical media objects.

The probability density function of the similarity score s, given that H_0 is true, is $p(s|H_0)$. $p(s|H_a)$ is defined accordingly. Let $\Gamma = \Gamma_{H_0} \cup \Gamma_{H_a}$ denote the set of all possible values of s, whereas Γ_{H_0} and Γ_{H_a} are two disjoint subsets of Γ. The null hypotheses is accepted if $s \in \Gamma_{H_0}$, otherwise it is rejected. P denotes probability. Then the FAR and FRR are defined as follows (also see figure 4.1):

Definition 4.1 (False Accept Rate (FAR)):
The FAR specifies the probability that two perceptually different images are identified as the same. In such a case, the similarity score is below the specified threshold. The FAR is specified as follows:

$$FAR = P(s \in \Gamma_{H_0}|H_1) = \int_{-\infty}^{T} p(s|H_1)ds. \tag{4.1}$$

Definition 4.2 (False Reject Rate (FRR)):
The FRR specifies the probability that two images which are perceptually the same are identified as different. In such a case, the similarity score is above the specified threshold. The FRR is specified as follows:

$$FRR = P(s \in \Gamma_{H_1}|H_0) = \int_{T}^{\infty} p(s|H_0)ds. \tag{4.2}$$

4.1.3 Receiver Operating Characteristic (ROC)

Suppose the integrals in 4.1 and 4.2 can be evaluated for any threshold T. Then the functions FAR(T) and FRR(T) give the FAR and respectively the FRR at the given threshold T. A Receiver Operating Characteristic (ROC) curve can then be obtained if the error rates are plotted against each other in a two-dimensional curve:

$$\text{ROC}(T) = (\text{FAR}(T), \text{FRR}(T)). \tag{4.3}$$

Figure 4.2 shows an example of a ROC curve. The FAR and FRR, as functions of T, are mapped as

$$ROC(T) = (FAR(T), FRR(T)) \mapsto \begin{cases} (1,0) & \text{as } T \mapsto -\infty, \\ (0,1) & \text{as } T \mapsto \infty. \end{cases} \tag{4.4}$$

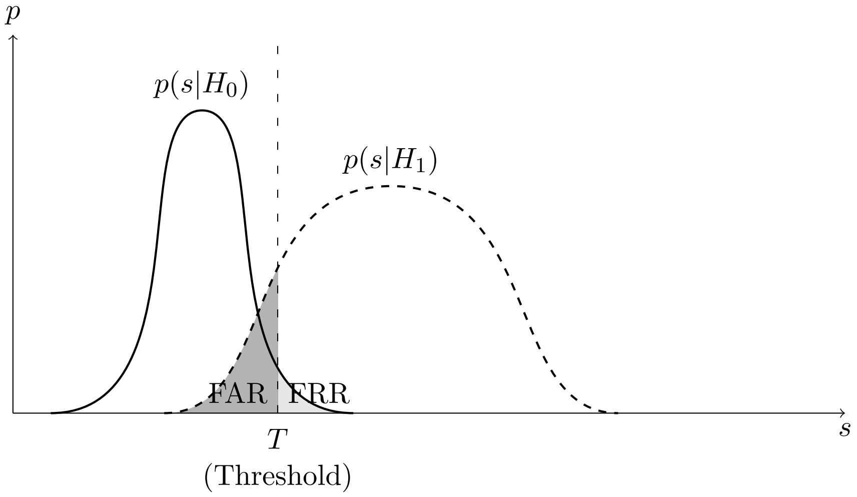


Figure 4.1: FAR and FRR.

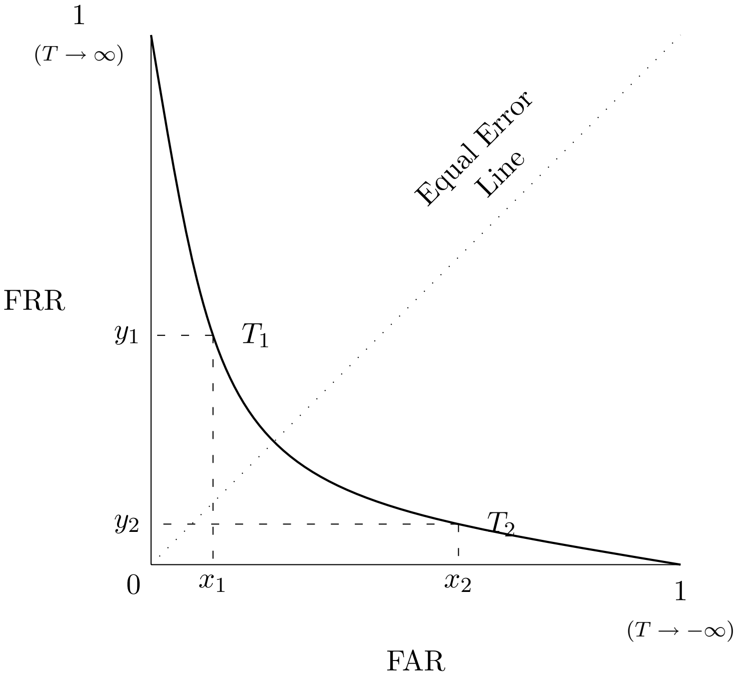


Figure 4.2: An example of a ROC curve. It expresses the trade-off between FRR and FAR. Cp. [2].

So when the threshold T is set low, the FRR is high and the FAR is low. Conversely, when T is high, the FRR is low and the FAR is high. A perceptual hash function can be operated using any theshold T, which defines a point on the ROC curve. This is the **operating point** of the perceptual hash function and it can be specified by choosing any one of T, FAR or FRR, with the other two then being implicitly defined.

For a perceptual hash algorithm it is especially hard to guarantee error rates that are low enough to be both fragile to perceptually distinct images (low FAR) and robust against perceptually insignificant image modifications (low FRR). Suppose that figure 4.2 depicts the ROC curve of a perceptual image hash function. Therefore the two possibilities the user could choose from then would be the following:

- The probability of a False Accept can be fixed at some (low) FAR $= x_1$. Consequently the probability of a False Reject is FRR $= y_1$.
- The probability of a False Reject can be fixed at some (low) FRR $= x_2$. Consequently the probability of a False Accept is FRR $= y_2$.

As previously outlined in this section there is always a trade-off between ceratain error types (e.g. FAR and FRR). The operating point of a perceptual hash function therefore has to be selected in accordance with the specific application it is used in. If a perceptual hash function is used for integrity verification for instance, the probability of falsely accepting an image that has been manipulated in a perceptual significant way should be as low as possible. Therefore, a high threshold is advisable. The best possible perceptual hash function would yield a point in the lower left corner (coordinate $(0, 0)$) of the ROC space. Such a function would have no false rejects (class 1 error) and no false accepts (class 2 error).

One has to be careful when reading and comparing ROC plots. According to [2] this is because *"... [there] does not appear to be a particular convention of the error trade-off as function of T [(threshold)] in biometrics; there are many variations but all boil down to the same thing."* Often different error types are plotted against each other. For instance, the False Accept Rate against the Correct Accept Rate. Such a curve is actually called "Detection Error Trade-off (DET) curve". Furthermore, the axes of such plots are frequently plotted on a logarithmic scale. Either both axes can be plotted on a logarithmic scale (log-log plot) or only one (semi-log plot). The logarithmic scaling is used to plot the interesting parts of a curve in a more detailed way.

While a ROC curve is a precise and complete specification of the performance of a single perceptual hash function, its real usefulness comes when comparing two perceptual hash functions. When comparing two perceptual hash functions it can hardly be decided unambiguously which one is better. That is because the performance of the perceptual hash functions depends

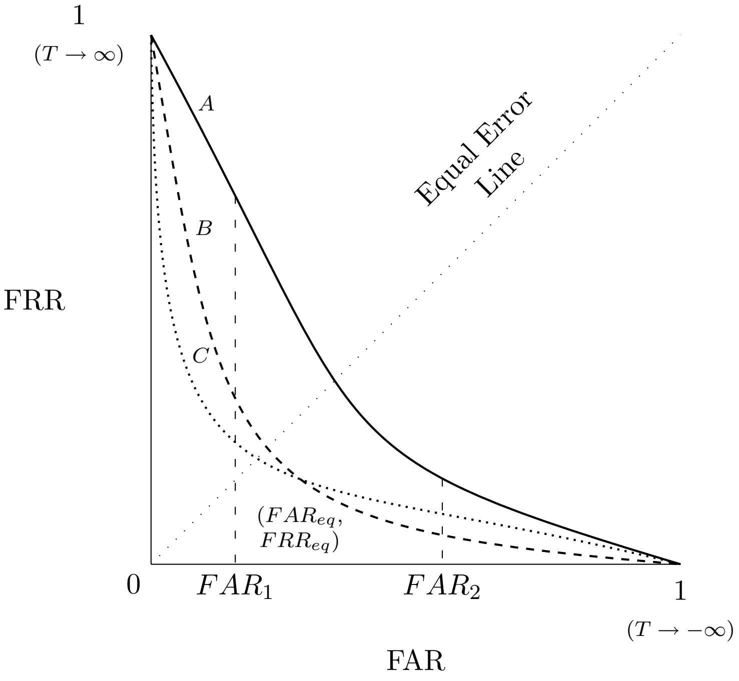


Figure 4.3: The actual operating point defines which perceptual hash function (A, B or C) is better. Cp. [2].

on their operating points. That is, it depends on the threshold T to judge the similarity scores. Figure 4.3 illustrates the ROC curves of the three perceptual hash functions A, B and C. It is clear that B and C are always better than A. That is because for every FAR that might be specified the FRR of B and C is lower. Likewise for every specified FRR their FAR is lower. If different operating points for the functions are used, it is possible for A of course to achieve a lower FRR **or** FAR than the other two.

When two ROC curves cross each other it can no longer be decided unequivocally which one is better. At operating point FAR_1 C is better than B. But at operating point FAR_2 the opposite is true. So if a low FAR is desired, (lower than FAR_{eq}) it is better to use C at e.g. operating point FAR_1. Likewise if a low FRR is desired (lower than FRR_{eq}) B is better suited at e.g. operating point FAR_2.

Several metrics can be derived from ROC curves to facilitate the comparison of different perceptual hash functions. Caution must be taken when using such metrics to compare different perceptual hash functions. These metrics are not able to reproduce the informations of a ROC curve completely. Nevertheless such metrics can be useful for summarizing a perceptual hash functions performance and comparing them (especially when the ROC curves do not cross).

Equal Error Rate

Definition 4.3 (Equal Error Rate (EER)):
The equal error operating point EE (the intersection of the Equal Error Line and a ROC curve) defines the Equal Error Rate (EER) of a perceptual hash algorithm. That is,

$$EER = FAR_{EE} = FRR_{EE}. \tag{4.5}$$

If the corresponding threshold is chosen, the probability that perceptually different objects are recognized as the same is as high as the probability that objects which are perceptually the same are not recognized as the same. In Figure 4.3 the hash function C has a lower EER than the hash functions A and B. Strictly speaking, the EER only depicts the performance at one given operating point. As can be seen in figure 4.3, ROC curves can cross over (B and C) and therefore a decision based on the EER would be erroneous. Hence the EER is only a very unreliable summary of a systems accuracy.

d-prime

The quality of a perceptual hash function can be measured by how much the probability density of authentic ($p(s|H_0)$) and not authentic attempts ($p(s|H_a)$) overlaps. See figure 4.1. The less these two probability densities overlap the better the authentic and not authentic attempts can be separated from each other.

Definition 4.4 (d-prime):
A measurement of the overlap, as suggested in [12], is d' (pronounced "d-prime"):

$$d' = \frac{\mu_m - \mu_n}{\sqrt{(\sigma_m^2 + \sigma_n^2)}}. \tag{4.6}$$

In this equation μ_m and σ_m are the mean and variance of the scores of authentic attempts, while μ_n and σ_n are the mean and variance of inauthentic attempts.

If d' equals 0 the probability densities of the authentic and not authentic attempts overlap completely. A possible cause for this is that the perceptual image hash function uses improper image features for the calculation of the image hash (e.g. image features which are almost the same for most images). Conversely, the bigger d', the less both probability densities overlap. As stressed in [2], d' can only be relied on for comparing two perceptual hash functions when there is a notable difference in performance. Two perceptual hash functions can have the same d' but exhibit substantial differences in performance, depending on the operating point chosen.

4.2 Metrics for Content Identification Systems

In this section the statistical error analysis of the previous section is extended to content identification systems (see Figure 2.2). Such systems have perceptual hash values of many (m) media objects stored in a database and, when a perceptual hash value is presented to them, determine which hash value, if any, matches.

An identification system compares the perceptual hash of an unidentified object to each of the hash values in the database. Though this can be seen as m "Yes/No" decisions (as in an integrity verification system). So for each hash value stored in the database a two-way hypothesis test is carried out:

H_0: The perceptual hash value of the unidentified object is in the database.

H_a: The perceptual hash value of unidentified the object is **not** in the database.

An ideal system will return m "No" (H_a) answers when the perceptual hash value of the unidentified object is not in the database. Consequently it will return a single "Yes" answer (H_0) and $m-1$ "No" answers if the perceptual hash value of the unidentified object is in the database. In practice a number of other situations will arise, yielding a variety of error conditions:

- More than one match, which might or might not include the correct media object (ambiguous answer).
- A single false match.
- No match despite a hash value of the media object is in the database.

In the rest of this section the FAR ($FAR(m)$) and FRR ($FRR(m)$) for identification systems using a database M with m hash values will be derived. The situation can be simplified by ignoring the case where multiple (correct or incorrect) hash values are matched. A media object is falsely accepted if one or more scores for incorrect hash values exceed the threshold. Under this assumption the chance of **correctly** rejecting a media object that is not in the database is

$$P(\text{correct reject}) = \prod_{i=1}^{m}(1 - \text{FAR}_i). \tag{4.7}$$

FAR_i denotes the separately measurable FARs for each media object in the database M. Although the FAR_i are non-identically but independently distributed random variables, FAR_i can be substituted with its expectation, further denoted as FAR (the overall system performance parameter), to obtain

$$P(\text{correct reject}) = (1 - \text{FAR})^m. \tag{4.8}$$

Definition 4.5 (FAR(m)):
Therefore the probability of a false accept can be defined as

$$FAR(m) = 1 - P(\textit{correct reject}) = 1 - (1 - FAR)^m. \tag{4.9}$$

This can be further simplified. If FAR $\ll 1$, then $(1 - \text{FAR})^m \approx 1 - m \cdot \text{FAR}$ holds true. Thus FAR(m) is approximately linear in m:

$$\text{FAR}(m) \approx m \cdot \text{FAR}. \tag{4.10}$$

A correct identification is considered to occur when the correct perceptual hash value of a media object is found in the database, no matter what happens with the other canidates. Thus,

$$P(\text{correct identification}) = 1 - \text{FRR}. \tag{4.11}$$

Definition 4.6 (FRR(m)):
The probability for a failed identification can then be derived as follows:

$$\begin{aligned} FRR(m) &= P(\textit{correct identification}) \\ &= 1 - (1 - FRR) \\ &= FRR. \end{aligned} \tag{4.12}$$

Therefore FAR(m) is independent of m. It equals the FRR of the perceptual hash function used in integrity verification mode.

To ease the understanding an example is illustrated. Given a perceptual hash function in integrity verification mode has a FAR of 0.01 (respectively 1%) and a FRR of 0.03 (respectively 3%). If such a function is used in content identification mode together with a database containing the hash values of 1000 media objects ($m = 1000$), then a FAR(m) of 0.99 (respectively 99%) and a FRR(m) of 0.03 (respectively 3%) are estimated.

4.2.1 Unambiguous Answers

If the previous assumptions can not be made the FAR(m) and FRR(m) can be refined, recognizing that an ambiguous answer is a failure of the system. An acceptance only occurs when exactly one candidate scores above the threshold, and is either correct or false depending on whether this candidate is the correct answer or some other media object.

First let us consider the case when the test subject is not in the database (not in M). A false accept occurs when exactly one database entry is falsely matched while all the others are rejected:

$$\begin{aligned} \mathrm{FAR}(m) &= \binom{m}{1} \cdot \mathrm{FAR} \cdot (1-\mathrm{FAR})^{m-1} \\ &= m \cdot \mathrm{FAR} \cdot (1-\mathrm{FAR})^{m-1}. \end{aligned} \tag{4.13}$$

If $\mathrm{FAR} \cdot m \ll 1$ then $(1-\mathrm{FAR})^{m-1} \approx 1$ holds true. So this reduces to

$$\mathrm{FAR}(m) \approx m \cdot \mathrm{FAR}. \tag{4.14}$$

The chance of clearly rejecting the test subject when it is not in the database is, as before, the probability of correctly rejecting all the entries in the database:

$$P(\text{correct reject}) = (1-\mathrm{FAR})^m. \tag{4.15}$$

The only remaining alternative is that an ambiguous answer is returned. That is, more than one candidate may exceed the threshold, giving an ambiguous candidate list (which might or might not include the correct media object). Its likelihood can be found as the remaining probability:

$$\begin{aligned} &P(\text{ambiguous answer (not authentic attempt)}) \\ &= 1 - P(\text{correct reject}) - \mathrm{FAR}(m) \\ &= 1 - (1-\mathrm{FAR})^m - m \cdot \mathrm{FAR} \cdot (1-\mathrm{FAR})^{m-1} \\ &= 1 - [(1-\mathrm{FAR}) - m \cdot \mathrm{FAR}](1-\mathrm{FAR})^{m-1} \\ &= 1 - [1 - (m+1) \cdot \mathrm{FAR}] \cdot (1-\mathrm{FAR})^{m-1}. \end{aligned} \tag{4.16}$$

Now the measures that can be derived from authentic attempts will be discussed. The chance of being **correctly** and **uniquely** identified (e.g. a media object stored in the database is being identified) is the probability of matching the correct perceptual hash, but none of the $m-1$ others:

$$P(\text{correct identification}) = (1-\mathrm{FRR}) \cdot (1-\mathrm{FAR})^{m-1}. \tag{4.17}$$

This is the **only case that counts as a non-ambiguous identification**. Therefore the FRR of the system can be defined as follows:

$$\begin{aligned} \mathrm{FRR}(m) &= 1 - P(\text{correct identification}) \\ &= 1 - (1-\mathrm{FRR}) \cdot (1-\mathrm{FAR})^{m-1}. \end{aligned} \tag{4.18}$$

This is higher than the FRR(m) derived in the previous section (equation 4.12). But when FAR $\cdot\, m \ll 1$, this reduces to

$$\mathrm{FRR}(m) \approx \mathrm{FRR}. \tag{4.19}$$

As outlined in [2], *"[a misidentification happens], when a **single** answer is returned, but it is the wrong answer. For this to happen, the correct record must be falsely rejected, while exactly one of the $m-1$ other records is falsely accepted (the rest being correctly rejected):"*

$$\begin{aligned} P(\text{misidentification}) &= \mathrm{FRR} \cdot \binom{m-1}{1} \cdot \mathrm{FAR} \cdot (1-\mathrm{FAR})^{m-2} \\ &= (m-1) \cdot \mathrm{FRR} \cdot \mathrm{FAR}(1-\mathrm{FAR})^{m-2}. \end{aligned} \tag{4.20}$$

The remaining alternative for an authentic attempt is to return an ambiguous answer. That is, more than one candidate may exceed the threshold, giving an ambiguous candidate list (which might or might not include the correct media object).

$$\begin{aligned} &P(\text{ambiguous answer (authentic attempt)}) \\ &= 1 - P(\text{correct identification}) - P(\text{misidentification}) \\ &= [(1-\mathrm{FRR})(1-\mathrm{FAR})^{m-1}] - [\mathrm{FRR}\cdot\mathrm{FAR}\cdot(m-1)(1-\mathrm{FAR})^{m-2}] \\ &= 1 - [1-\mathrm{FRR}-\mathrm{FAR}+m\cdot\mathrm{FRR}\cdot\mathrm{FAR}]\cdot(1-\mathrm{FAR})^{m-2}. \end{aligned} \tag{4.21}$$

For an identification system it is not only important that the FAR(m) and the FRR(m) are low but also the probability of ambiguous answers must be low. [2] lists four possibilites that ambiguous answers can be dealt with:

1. Running an exception procedure. A human supervisor screens the list of possible candidates for instance. A human supervisor is the most time-consuming and expensive possibility. Otherwise the quality of the decision of a human supervisor is usually superior to the quality of the descision made by perceptual hash functions.

2. Ambiguous answers can be considered as rejects, thus increasing the FRR(m) above its value in equation 4.19. When searching for content that is relevant under criminal law aspects, a small FRR is desired. Thus, this possiblity is suboptimal for such an application.

3. Passing the possible answers on to some other identification system. The robustness and discriminative abilities of perceptual hash functions differ vastly with regard to different content changes. Some perceptual hash function are optimized to be robust against e.g. rotation operations whereas others are optimized to be robust against e.g.

JPEG compression. Therefore by combining perceptual hash functions that are optimized for different operations an identification system can be enhanced.

4. Accept the highest-scoring candidate. This approach is easy to implement and adequate for certain applications (e.g. to check if an image already exists in an image database).

Chapter 5

Rihamark Benchmarking Framework

5.1 Review of Related Work and Open Issues

For watermarking, different benchmarks have been developed and are partially well-established (e.g. Stirmark[1] , CheckMark[2] , Optimark[3] or Certimark[4]). The design and application flow of perceptual hash functions differs vastly from digital watermarking functions. Therefore watermarking benchmarks can not be used to evaluate perceptual hash functions.[45] Not much research has been published dealing with the benchmarking of perceptual hash functions. In [45] a *"novel benchmark platform for perceptual hashing algorithms"*, called Perceptual Hashing Algorithms Benchmark Suite (PHABS), was published. One of the authors, Hui Zhang, covered PHABS in a more in-depth way in [41]. PHABS is written in C++. Neither of the previously mentioned publications specifies which operating sytems are supported by PHABS. The author of this thesis was not able to get in contact with the authors behind PHABS, to get a compiled version of PHABS or to get hold of its source code. As of the date of the writing of this thesis, no other references to PHABS could be found on the World Wide Web. Consequently it must be assumed that PHABS has been abandoned and is not publicly available.

As a consequence, the development of a benchmarking framework for perceptual image hash functions had to be started from scratch. The developed benchmarking framework is named **Rihamark**.

[1] Homepage: `http://www.cl.cam.ac.uk/~fapp2/watermarking/stirmark/`

[2] Homepage: `http://watermarking.unige.ch/Checkmark/index.html`

[3] Homepage: `http://poseidon.csd.auth.gr/optimark/`

[4] Homepage: `http://www.certimark.org/`

5.2 Design Overview

This section outlines the design and implementation of Rihamark. The primary design goals of Rihamark can be summarized as follows. Rihamark should provide the ability...

- for a user to add his own perceptual hash functions, attack functions or analyzer functions.
- to define and execute a test plan consisting of an arbitrary number of perceptual hash functions or attack functions.
- to monitor the execution status of a test plan.
- to retrieve the results of a test plan.
- to present the results of a test plan to the user and analyze them in a statistical way.
- to be used on any major operating system (Microsoft Windows and Unix-based operating systems).
- to be used without requiring commercial third party programs (e.g. Matlab).

Based on these goals the design of Rihamark was derived. Rihamark is written in the Java programming language. The Java runtime environment was chosen because it is available for most of the major operating systems and a lot of libraries are freely available for non-commercial use. Furthermore, it provides mechanisms to call methods written in other programming languages (e.g. C/C++ using the JNI). Rihamark consists of three main components. Each of these main components resides in its own Java package:

Package `rmk.core`: The Rihamark Core is the actual benchmarking framework. It manages the data structures necessary (e.g. a test plan) for the benchmarking of perceptual image hash functions and executes the actual benchmarking. Furthermore it is responsible for the management of the plugins.

Package `rmk.gui`: The Rihamark Graphical User Interface (GUI) is an implementation of a user interface for the Rihamark Core. Basically it would also be possible to write e.g. a command line interface for the Rihamark Core. A GUI was preferred because it allows a user to easily create and manipulate test plans. Additionally, Rihamark offers analyzer plugins which visualize the results of a test plan using a GUI. A command line interface would only be able to save those visualizations directly to the hard disc.

Package `rmk.plugins`: This package includes the default plugins of Rihamark. There are `Attack`[5], `Algorithm` and `Analyzer` plugins available. A user can extend the functionality of Rihamark by writing his own plugins.

There are three other additional packages. Namely the `rmk.SPArguments`, the `rmk.SPInterfaces` and the `rmk.SPMenu` package. These packages provide classes and interfaces that must be used by plugins or user interfaces to collaborate with the Rihamark core package. The following sections 5.3, 5.4 and 5.5 discuss the three major packages of Rihamark in more detail.

5.3 Rihamark Core

The Rihamark Core manages the data structures neccessary for benchmarking. A class diagram of the Rihamark Core is shown in figure 5.1. The class diagram is subtotal. It encompasses only the parts neccessary for the following discussion of the Rihamark Core. The topmost entity is a test plan (class `TestPlan`). Only one `TestPlan` at a time can be managed. A `TestPlan` consists of an arbitrary number of tests (class `Test`). Such a test can, in turn, encompass an arbitrary number of algorithms (abstract class `Algorithm`) and attacks (abstract class `Attacks`). Various other classes of the Rihamark Core implement actions on `TestPlan` and `Test` objects. The dispatcher (class `Dispatcher`) is such an example.

5.3.1 The TestPlan Class

The `TestPlan` class is influenced by various member variables. These member variables are reflected one by one by the options the Rihamark GUI offers. Subsequently the names of the member variables are omitted and only the names of the options the Rihamark GUI offers are given. These member variables have also been partially omitted in the class diagram in figure 5.1. If the **"Save results of test plan"** option is set, the results of the test plan are saved after the test plan has been finished successfully. Regardless of whether this option is set, the results are always printed out to the standard output stream of the operating system. **"Save results as"** determines the format that is used to save the results. At the moment it is only supported to save the results to a text file. The option **"File path"** can be used to modify the path for saving the text file. It is also possible to accumulate the results of multiple test plan runs. However, as this is not the expected behaviour the option **"Delete results of previous test plan runs. . . "** is enabled by default.

[5] Although these plugins can also just be used for modifying an image they are called attack plugins.

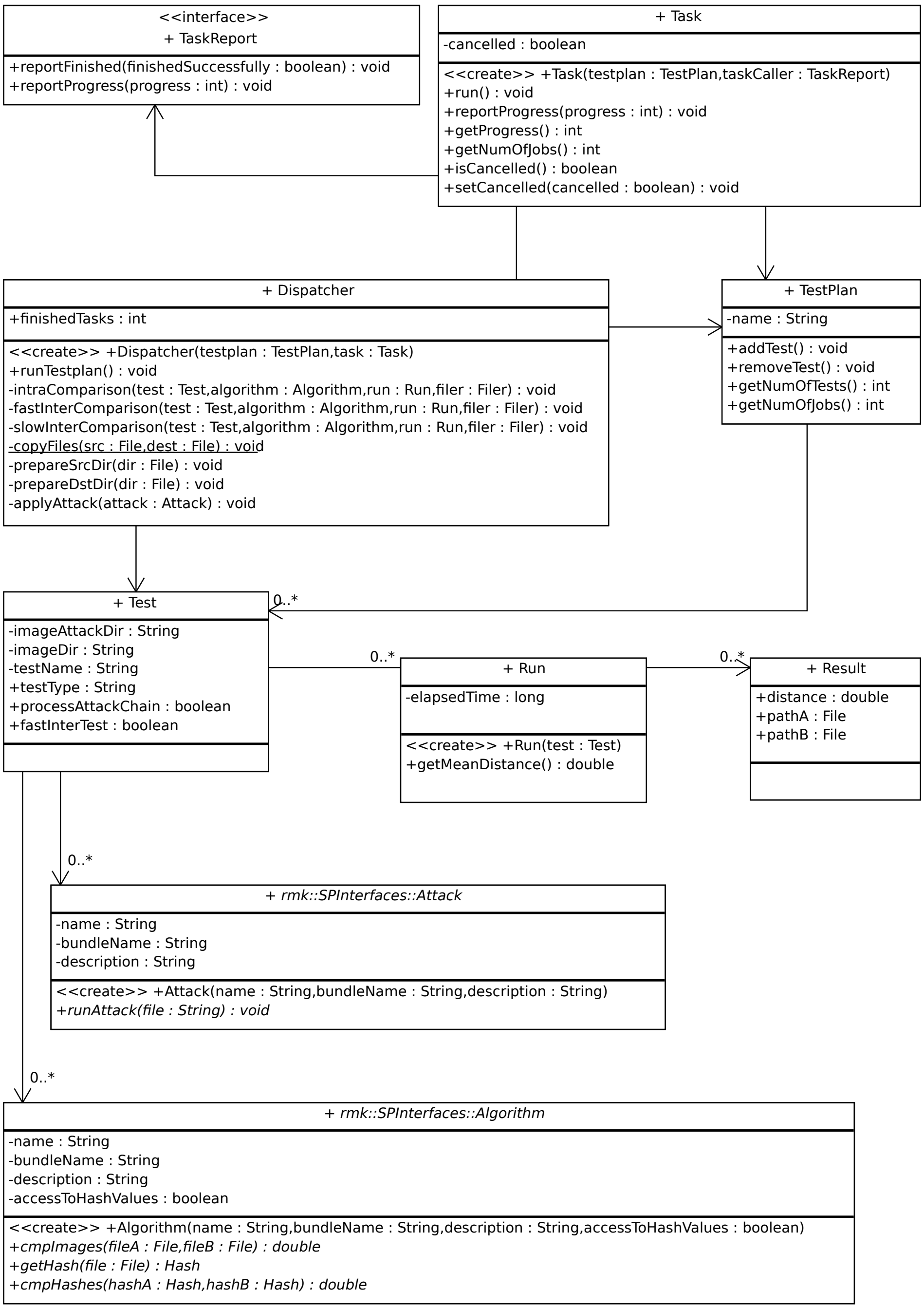

<<interface>>
+ TaskReport
+reportFinished(finishedSuccessfully : boolean) : void
+reportProgress(progress : int) : void
+ Task
-cancelled : boolean
<<create>> +Task(testplan : TestPlan,taskCaller : TaskReport)
+run() : void
+reportProgress(progress : int) : void
+getProgress() : int
+getNumOfJobs() : int
+isCancelled() : boolean
+setCancelled(cancelled : boolean) : void
+ Dispatcher
+finishedTasks : int
<<create>> +Dispatcher(testplan : TestPlan,task : Task)
+runTestplan() : void
-intraComparison(test : Test,algorithm : Algorithm,run : Run,filer : Filer) : void
-fastInterComparison(test : Test,algorithm : Algorithm,run : Run,filer : Filer) : void
-slowInterComparison(test : Test,algorithm : Algorithm,run : Run,filer : Filer) : void
-copyFiles(src : File,dest : File) : void
-prepareSrcDir(dir : File) : void
-prepareDstDir(dir : File) : void
-applyAttack(attack : Attack) : void
+ TestPlan
-name : String
+addTest() : void
+removeTest() : void
+getNumOfTests() : int
+getNumOfJobs() : int
+ Test
-imageAttackDir : String
-imageDir : String
-testName : String
+testType : String
+processAttackChain : boolean
+fastInterTest : boolean
0..*
0..*
+ Run
-elapsedTime : long
<<create>> +Run(test : Test)
+getMeanDistance() : double
0..*
+ Result
+distance : double
+pathA : File
+pathB : File
0..*
+ rmk::SPInterfaces::Attack
-name : String
-bundleName : String
-description : String
<<create>> +Attack(name : String,bundleName : String,description : String)
+runAttack(file : String) : void
0..*
+ rmk::SPInterfaces::Algorithm
-name : String
-bundleName : String
-description : String
-accessToHashValues : boolean
<<create>> +Algorithm(name : String,bundleName : String,description : String,accessToHashValues : boolean)
+cmpImages(fileA : File,fileB : File) : double
+getHash(file : File) : Hash
+cmpHashes(hashA : Hash,hashB : Hash) : double

Figure 5.1: UML class diagram of the package `rmk.core`. The class diagram is greatly simplified.

5.3.2 The Test Class

The `Test` class has numerous important options that affect the benchmarking process. The **"Image directory"** contains the images that shall be used for benchmarking. The content of this directory is not modified in any way. The Rihamark Core recursively copies all supported image files (see table 5.1) found in this directory to the **"Attack image directory"**. Already existing files in the attack image directory will be overwritten without prompting the user. If the image directory or the attack image directory does not exist it will be created. The **"Test name"** option can be used to assign a custom name to the test. When saving a test plan to an Extensible Markup Language (XML) file this option can ease the manual interpretation of such an XML file. Therefore it is suggested to assign a name that outlines the purpose of a test (e.g. intra test with rotation and flip attack chain items). A `Test` has a so-called attack chain. An attack chain consists of an arbitrary number of `Attack` objects. The `Dispatcher` applies the attacks sequentially to each image in the attack image directory. Thus, a user is able to change images by an arbitrary combination of image operations (e.g. rotation by 10°, scaling by 150% and JPEG compression with a quality setting of 70).

The user has to choose which type a test should have. A test can be of **"Test Type"** intra or inter. Suppose that $I = \{a, b, c, d\}$ denotes the set of the images in the image directory and $I' = \{a', b', c'\}$ denotes the set of the images in the attack image directory. When performing an intra test the distance scores of $(a, a'), (b, b'), (c, c'), (d, d')$ are calculated. Thus, each image in the image directory is copied to the attack image directory. Then the attack chain is applied to each image in the attack Image directory. Afterwards, the distance between each (original) image in the image directory and its counterpart in the attack image directory is calculated. When performing an inter test the images are copied and changed as before. But subsequently the procedure for the calculation of the distance scores changes. Instead of 4 distance scores, the 6 distance scores of $(a, b'), (a, c'), (a, d'), (b, c'), (b, d'), (c, d')$ are calculated. Thus, each image in the image directory is compared with all the other images in the attack image directory. The only exception is that an image gets never compared with itself.

If the option **"Process attack chain"** is not set, the images in the image directory are not copied to the attack image directory. The purpose of this option is to be able to use an image set that has been changed by any other means than one of the Rihamark attack plugins. The option **"Use hash values for inter tests"** (member variable `fastInterTest`) is only relevant when performing an inter test. An inter test can be performed at three different rates. The slow rate is used if the Rihamark plugin of the hash function is only able to return a distance or similarity value for two given files to the Rihamark Core. This has the advantage that neither the plugin, nor the Rihamark Core have to support the data type used for the hash values of a specific hash function. The fast rate is used if the plugin of the hash function is able to return the actual hash value to the Rihamark Core. This procedure offers an enormous speed gain. The fastest rate is in action if the images in the attack image directory have not been changed. That is, no attacks have been applied and thus the images in the image directory are exactly the same as the images in the attack image directory ($I = I'$). The possible speed gains can be outlined by an example. Suppose an image set consisting of 1000 images is used in an inter test. At the slow rate, a hash function would have to hash $\binom{1000}{2} = 4950$, at the fast rate $1000 \cdot 2 = 2000$ and at the fastest rate 1000 images.

5.3.3 The Filer Class

The `Filer` class is invoked by the `Dispatcher` class. It is responsible for scanning the image directory and creating a file list of the images therein. There are several restrictions regarding the file extensions (and formats) of image files. Invalid file extensions are simply ignored during a benchmark run. Only lower case file extensions are valid. Table 5.1 lists all valid file extensions.

File extension	Image format
.bmp	Windows Bitmap
.gif	Graphics Interchange Format
.jpg	Joint Photographic Experts Group (JPEG)
.png	Portable Network Graphics (PNG)
.tif	Tagged Image File Format

Table 5.1: Supported image formats and file extensions of the Rihamark benchmarking framework.

5.3.4 The Dispatcher Class

The actual benchmarking is carried out by the `Dispatcher` class. It executes the actions described in a `TestPlan`. The benchmarking results are saved on a per run basis. That is, a `Run` saves the outcome of a benchmarking run collected when testing an algorithm for a specific image operation / attack chain (e.g. scaling together with rotation). A `Run` object offers the following:

- The time it took the algorithm to complete the requested hashing operations is managed.
- A method is offered to get the mean distance of the calculated distance scores.

- A list of results (class Result, see below for more information), is managed.

A Result consists of the file names of the images that were hashed and the distance score the perceptual hash algorithm assigned to them. If a perceptual hash function uses a similarity score to compare two perceptual hash values the similarity score has to be converted to a distance score:

$$\text{distance score} = 1 - \text{similarity score}. \tag{5.1}$$

5.3.5 Miscellaneous Classes

The Conservator class offers methods to save or load a TestPlan to or from an XML file. To achieve this kind of serialization and deserialization the Simple Java library is used. The raw results of a benchmarking run can be printed to the standard output file descriptor or saved to text file using the Printer class. This makes it possible for a user to process the results of a benchmarking run with his own programm of choice (e.g Gnuplot or R).

The Rihamark Core provides a logging facility which other components, like the default rotation attack plugin or the Rihamark GUI, make use of. Five verbosity levels are available, namely ERR, WARN, INFO, DEBUG1 and DEBUG2. The logging facility is implemented by the Debug class.

5.3.6 Communication with User Interfaces

If a user interface wants to execute a TestPlan it has to create a new Task object. The constructor of a Task object takes two arguments. A TestPlan object and a TaskReport object. The latter is an interface which each user interface of the Rihamark Core has to implement (see listing B.3). The Core uses the methods of this interface to signal various events to the user interface. To start the execution of a TestPlan by the Dispatcher a user interface has to invoke the start() method on a Task object. The Task object then creates a new Dispatcher object and starts its execution in a worker thread. After this the Task object returns immediately. The Task class also offers methods which a user interface can invoke to get the total number of jobs the Dispatcher has to execute, how many jobs have been completed and to signal that the user wants the Dispatcher to abort the execution of the TestPlan. A job is defined as an Attack or Algorithm that is part of a TestPlan.

5.3.7 Plugin Architecture

Rihamarks design goal was to be versatile and flexible. In order to achieve this goal Attack, Algorithm and Analyzer objects are realized as plugins (service providers). Rihamark comes with a set of default service providers. They are discussed in section 5.4.

Definition 5.1 (service provider):
A service provider implements a Service Provider Interface (SPI).

The goal is to be able to add new service providers to an application – in order to extend its functionality – without modifying the original source code. A colloquial term for service provider is "plugin".

Definition 5.2 (Service Provider Interface (SPI)):
A SPI is a set of one or more public interfaces that a service provider has to implement in order to be useable by a service.

Definition 5.3 (service):
A service sits in front of all service providers. The service loads available service providers on behalf of the service user.

The plugin architecture is visualized in figure 5.2.

A service provider for Rihamark needs a facility to get inputs from the user. To outline the situation, let us take a closer look at one of the `Attack` service providers that is part of the Rihamark default plugin package (see section 5.4). The class `Rotate`, as the name implies, rotates an image. Such an image operation is always characterized by a certain set of properties. The `Rotate` class has the following three properties: `angle`, `interpolation` and `enlarge`. These three properties specify how exactly an image is going to be rotated. Because the service provider can't make any assumptions regarding the user interface (e.g. one could interface with Rihamark using a GUI or a command line interface) the Rihamark Core offers a facility for service providers to specify which user interface elements they need. A service provider has to initialize possible user interface elements in his constructor. Listing B.4 shows the constructor of the `Rotate` service provider. Figure 5.3 shows how the Rihamark GUI implements the requested user interface. An `Attack` or `Algorithm` service provider can use the following user interface controls to interface with the user:

`CheckBox`: The `value` member variable of this class stores a `boolean`. The Rihamark GUI implements this argument using a `JCheckBox` Swing control.

`ComboBox`: The `value` member variable of this class stores a `String`. The `String[]` member variable `items` specifies the string values the user can choose from. The Rihamark GUI implements this argument using a `JComboBox` Swing control.

`Label`: The `value` member variable of this class stores a `String`. The user can not modify the member variable `value`. The class is supposed

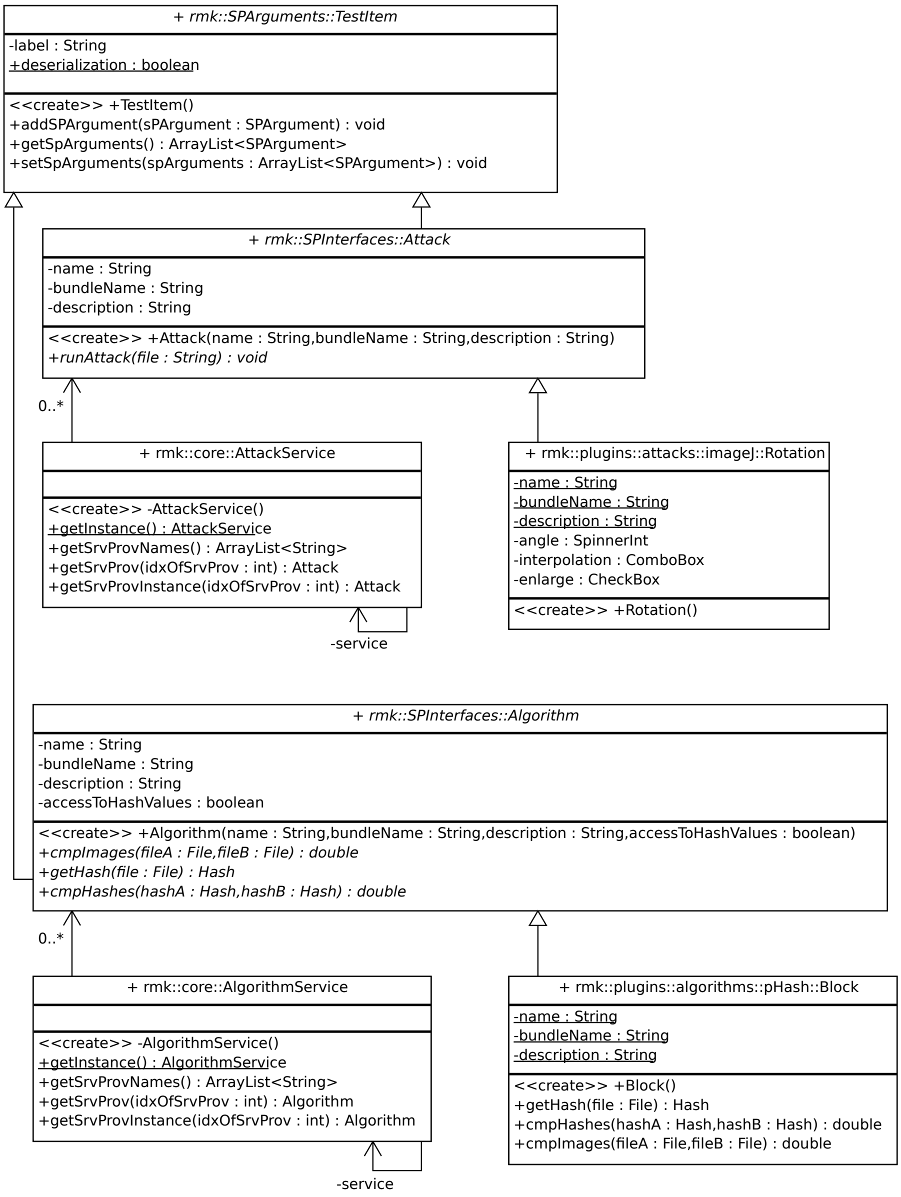
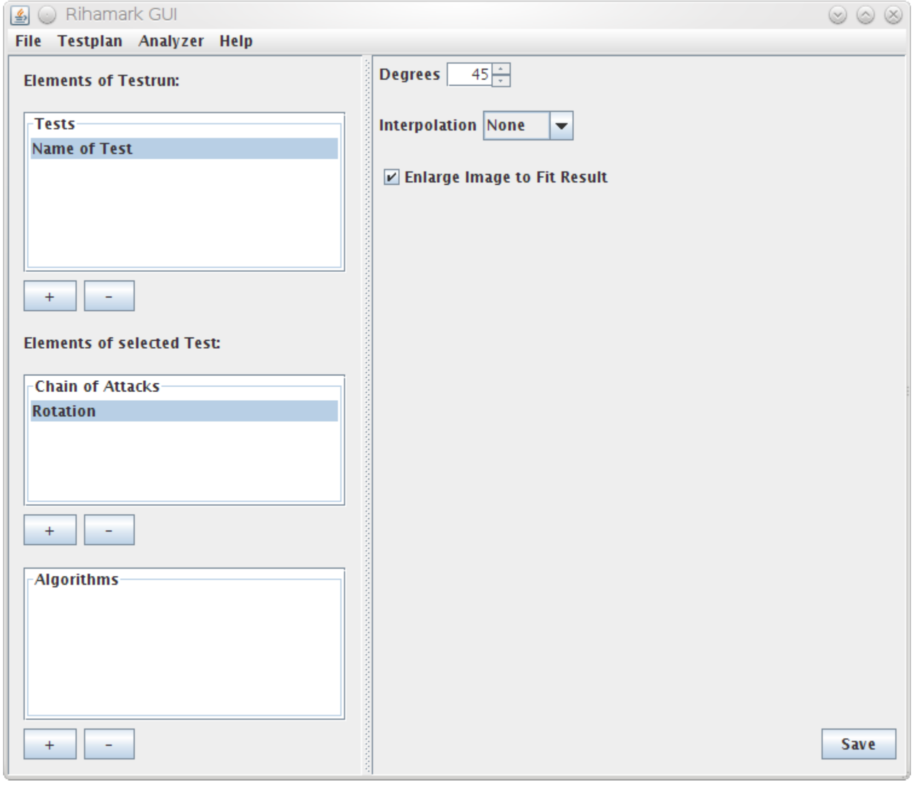

+ rmk::SPArguments::TestItem
-label : String
+deserialization : boolean
<<create>> +TestItem()
+addSPArgument(sPArgument : SPArgument) : void
+getSpArguments() : ArrayList<SPArgument>
+setSpArguments(spArguments : ArrayList<SPArgument>) : void
+ rmk::SPInterfaces::Attack
-name : String
-bundleName : String
-description : String
<<create>> +Attack(name : String,bundleName : String,description : String)
+runAttack(file : String) : void
0..*
+ rmk::core::AttackService
<<create>> -AttackService()
+getInstance() : AttackService
+getSrvProvNames() : ArrayList<String>
+getSrvProv(idxOfSrvProv : int) : Attack
+getSrvProvInstance(idxOfSrvProv : int) : Attack
-service
+ rmk::plugins::attacks::imageJ::Rotation
-name : String
-bundleName : String
-description : String
-angle : SpinnerInt
-interpolation : ComboBox
-enlarge : CheckBox
<<create>> +Rotation()
+ rmk::SPInterfaces::Algorithm
-name : String
-bundleName : String
-description : String
-accessToHashValues : boolean
<<create>> +Algorithm(name : String,bundleName : String,description : String,accessToHashValues : boolean)
+cmpImages(fileA : File,fileB : File) : double
+getHash(file : File) : Hash
+cmpHashes(hashA : Hash,hashB : Hash) : double
0..*
+ rmk::core::AlgorithmService
<<create>> -AlgorithmService()
+getInstance() : AlgorithmService
+getSrvProvNames() : ArrayList<String>
+getSrvProv(idxOfSrvProv : int) : Algorithm
+getSrvProvInstance(idxOfSrvProv : int) : Algorithm
-service
+ rmk::plugins::algorithms::pHash::Block
-name : String
-bundleName : String
-description : String
<<create>> +Block()
+getHash(file : File) : Hash
+cmpHashes(hashA : Hash,hashB : Hash) : double
+cmpImages(fileA : File,fileB : File) : double

Figure 5.2: UML class diagram of the plugin architecture. The class diagram is greatly simplified and the classes concerning the `Analyzer` plugins are omitted.



Degrees 45

Interpolation None

☑ Enlarge Image to Fit Result

Figure 5.3: Screenshot that shows how the Rihamark GUI renders the user interface of the `Rotation` plugin.

to be used as a solitary label. The Rihamark GUI implements this argument using a `JLabel` Swing control.

`SpinnerDbl`: The `value` member variable of this class stores a `double`. Furthermore, there are three constraint variables. `double minimum` is the minimal allowed `value` whereas `double maximum` is the maximum allowed `value`. The `double stepSize` specifies the step size of `value`. The Rihamark GUI implements this argument using a `JSPinner` Swing control.

`SpinnerInt`: The `value` member variable of this class stores an `int`. Furthermore, there are three constraint variables. The `int minimum` is the minimal allowed `value` whereas `int maximum` is the maximum allowed `value`. The `int stepSize` specifies the step size of `value`. The Rihamark GUI implements this argument using a `JSPinner` Swing control.

5.4 Default Plugins

5.4.1 Attack Plugins

Rihamark offers numerous default attack plugins. They are located in the package `rmk.plugins.attacks`. The plugins actually use ImageJ to execute the requested operations. The following plugins are available:

`Flipper`: This attack flips an image horizontally or vertically. The user has to choose the axis.

`Gaussian Blur`: This attack blurs the image using a Gaussian filter with a user-specified sigma.

`JpegMangler`: The JPEG compression attack is implemented as follows. First the image is converted to the JPEG image file format using the specified quality setting. The resulting image file is saved to the temporary system directory. Then the JPEG image is read by the attack plugin and finally it is saved to the attack image directory using the original image file format.

`PngMangler`: The Portable Network Graphics (PNG) compression attack is implemented like the JPEG compression attack. This attack plugin has no options.

`Resize`: Resizes an image to the given dimension. The user can set the desired width and height and select which interpolation should be used. He can also choose if the width or height should be adjusted proportionally.

`Rotation`: Rotates an images by the given angle. Furthermore, the user can choose which interpolation to use and if the canvas of the resulting image should be enlarged if neccessary.

`Scale`: Scales the image using the given factors for width and height. The user can also choose which interpolation to use.

The attack plugins use the file extensions to decide which image file format to use for saving a changed image. Lossy compression formats like JPEG are perceived as one self-contained attack when benchmarking perceptual image hash functions. Therefore the use of lossy compressed images is discouraged.

5.4.2 Algorithm Plugins

Rihamark offer numerous default algorithm plugins. They are located in the package `rmk.plugins.algorithms`. All the perceptual image hash functions of pHash are currently supported. See section 3.2 for a summary of the implemented functions.

5.4.3 Analyzer Plugins

Rihamark offer numerous default analyzer plugins. They are located in the package `rmk.plugins.analyzers`. The analyzer plugins offer methods to analyze the results of a test plan in a statistical way. Some of them expect the test plan results to be in a certain format. For the visualization of the test results the JFreeChart java library is utilized. Charts can be exported into PNG, Scalable Vector Graphics (SVG) or Portable Document Format (PDF). The export to SVG is achieved by using the Batik Java library. The iText Java library is used for PDF export. Currently the following charts can be created using the default analyzer plugins:

Score distribution chart: The score distribution uses a scatter plot to visualize the distribution of the distance scores. It can be used to judge the discrimination ability of a hash function. It can also be used to visualize other performance indicators of hash functions (e.g. the robustness of perceptual hash functions). For an example of such a chart see figure A.2.

ROC chart: To create a ROC chart (see figure 5.4), the analyzer needs the results of a test plan to be in a special format. The first two tests of the test plan are used to derive one ore more ROC curves. The first test has to contain one or more intra (authentic) tests, whereas the second test has to contain one or more inter (not authentic) tests.

Speed comparison chart: During the execution of a test plan, the dispatcher measures how much time each perceptual hash function needs to calculate all the required image hashes for a specific test run. A speed comparison chart (see figure A.1 for an example) visualizes the time. The chart can be used to evaluate the performance in terms of speed of the actual implementation of a perceptual hash function.

Effect of attack chart: This function creates a chart to visualize the effect of an image operation / attack when its parameters are varied. A line chart is used for visualization. The test plan should consist of multiple tests. Each test should encompass the same algorithms. The attack parameters should be varied. See figure A.9 for an example.

5.5 Rihamark GUI

The Rihamark GUI is an implementation of a GUI for the Rihamark Core. It is written in Java and uses Swing. It is responsible for creating a test plan and to represent the contents of a test plan to the user. Furthermore, it offers a GUI that enables a user to manipulate all sorts of properties of a test plan. To see how the GUI implements the user interface for the various options of the service providers please see section 5.3.7.

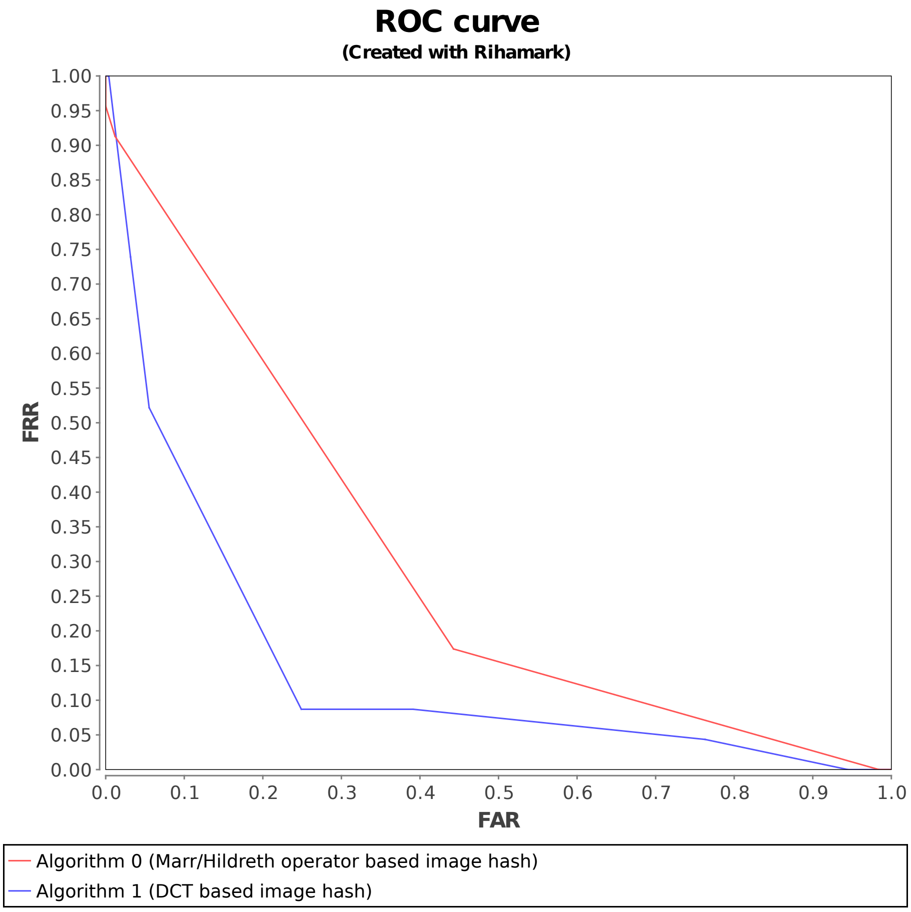


Figure 5.4: ROC chart created with Rihamark.

Chapter 6

Benchmark Results

Four different perceptual image hash functions were benchmarked using the Rihamark benchmarking framework. All the functions are implemented by pHash. The block mean value based perceptual image hash function was contributed by the author of this thesis to pHash. The four perceptual image hash functions were previously discussed in section 3.1. Their implementaion in pHash was discussed in section 3.2. It is important to remember that certain properties of a perceptual image hash function (e.g. speed) can vary heavily depending on their specific implementation. But there are other properties where this is not so obvious. When implementing certain perceptual image hash functions, one has to make certain assumptions because of the lack of detail provided by some of the "specifications" of these functions. The following topics were identified for benchmarking.

Speed: The sheer speed was subject of this benchmark.

Inter score distribution: The inter score (not authentic) score distributions were benchmarked using different image sets. The first set consists of rather dissimilar images, whereas the second set consists of rather similar ones.

Intra score distribution: The intra score (authentic) score distributions were benchmarked using different operations.

Different image sets were used for the benchmarks. All the images were obtained from Wikimedia Commons[1]. The images are so-called "quality images". Quality images are images which meet certain quality standards (which are mostly technical in nature) and which are valuable for Wikimedia projects. The first set, hereinafter referred to as the "**event image set**", consists of images with very different motifs. The image set consists of 47 images. The images of the set depict various events. The mean dimension

[1] Homepage: `http://commons.wikimedia.org/`

of the images is 2874 x 2260 pixels. Their mean file size is 3.19MiB. The total file size of the 47 images is 149.77MiB. This image set was taken from the Wikimedia Commons web site "Quality images/Subject/Events"[2]. The second image set, hereinafter referred to as "**duck image set**", mainly consists of photographs that show ducks swimming in the water. It consists of 45 images. The mean dimension of the images is 2732 x 1802 pixels. The mean file size is 2.75MiB. The total file size of the 45 images is 123.79MiB. The image set is a selection of images from the Wikimedia Commons web site "Quality images/Subject/Animals/Birds"[3]. The third image set, hereinafter referred to as "**chaos image set**", consists of images with varying motifs. The images were taken from various quality image sets from Wikimedia Commons. It consists of 45 images. The mean dimension of the images is 2502 x 2200 pixels. The mean file size is 2.44MiB. The total file size of the 45 images is 109.90MiB. The fourth and last image set is a subset of the chaos image set. Hereinafter it will be referred to as the "**small chaos image set**". Three images were taken from the chaos image set to form this image set. The mean dimension of the images is 3003 x 2222 pixels. The mean file size is 3.51MiB. The total file size of the 3 images is 10.54MiB.

The following sections present and discuss the results of the benchmarks. Each topic is adressed in a separate section. All perceptual image hash functions were configured to use their default parameters (see table 6.2). Table 6.1 shows the hardware and software of the system on which the benchmarking was carried out.

CPU	Intel Core 2 Duo T9300 (2.50GHz)
RAM	4096MiB
HDD	Seagate Momentus 5400.4 250GB (SATA, 3Gb/s), Model Nr.: ST9250827AS
OS	32-Bit GNU/Linux distribution

Table 6.1: Hard- and software of the system used for benchmarking.

6.1 Speed

The speed of a perceptual image hash function is especially important when a great number of images needs to be hashed and processed. This is e.g. the case when searching the World Wide Web for copyright infringements.

For benchmarking the event image set was used. The assembled test plan consisted of one test item. The test type option was set to "Intra".

[2] Web site of the Wikimedia Commons quality images of events: `http://commons.wikimedia.org/wiki/Commons:Quality_images/Subject/Events`.

[3] Web site of the Wikimedia Commons quality images of birds: `http://commons.wikimedia.org/wiki/Commons:Quality_images/Subject/Animals/Birds`.

DCT based hash	
No parameters available.	n/a
Marr-Hildreth operator based hash	
Wavelet scale factor	2
Scale factor level	1
Radial variance based hash	
Sigma (radius) of the gaussian filter	1
Gamma correction	1
Number of angles to consider	180
Block mean value based image hash	
Method	1
Preset size X	256
Block size X	16

Table 6.2: pHash default parameters.

Therefore each perceptual image hash function had to hash 47 × 2 = 94 images. An attack of type "Empty" was added to the attack chain. Therefore Rihamark copied the images from the "Image Directory" to the "Attack Image Directory" without changing them.

The result of the benchmark is shown in figure A.1. The results are summarized in table 6.3. The newly implemented block mean value based perceptual image hash function is the fastest hash function. It needs **58** seconds to hash 94 images. The second fastest hash function, with **118** seconds, is the radial variance based hash function. Far behind are the Marr-Hildreth operator based (**343** seconds) and the DCT based (**911** seconds) hash functions. The great differences in speed can be explained by the fact that the former two hash functions only use pixel operations for feature extraction when calculating the hash. By contrast, the latter two use computationally more expensive convolution/correlation operations.

	DCT	MH	Radial	BMB
Total time (sec.)	911	343	118	**58**
avg. sec. per image	9.7	3.6	1.3	**0.6**
MiB/sec.	0.33	0.87	2.54	**5.16**

Table 6.3: Statistical results of the speed benchmark. The best result in each category is printed in bold.

6.2 Inter Score Distribution

The inter score distribution can be used to measure and judge the discriminative capabilities of a perceptual hash function. When comparing two different images a perfect perceptual hash function would always yield a distance (or similarity score) of 0.5. An interesting question is whether the score distribution depends on the used images. When using very similar images it may be more difficult for a perceptual image hash function to achieve the "perfect" distance of 0.5 for every comparison. A thousand images, all of them depicting snow-covered mountains, can be considered as such a similar image set. Therefore the inter score distribution was benchmarked using two different image sets. The first set was the chaos image set, whereas the second set was the duck image set. Each perceptual image hash function had to calculate $\binom{45}{2} = 990$ hash values per image set.

The results of the intra tests are depicted in figures A.2 – A.5 and summarized in table 6.4. Both the figures and the summary apparently indicate that the Marr-Hildreth operator based image hash has by far the most discriminative abilities. The DCT based image hash performs as second best. Figure A.2 suggests that when using the DCT based hash function, specific distance scores occur very often (e.g. 0.531 or 0.469). The radial variance based image hash is on a par with the block mean value based image hash. Another interesting conclusion can be drawn from this benchmark. The composition of the image set does not significantly influence the performance of the benchmarked perceptual image hash function.

	DCT	MH	Radial	BMB
Run 0 (chaos set):				
Mean dist.	**0.501**	**0.499**	0.565	0.482
Max. dist.	0.688	**0.578**	0.812	0.812
Min. dist.	0.250	**0.408**	0.135	0.109
Run 1 (duck set):				
Mean dist.	0.496	**0.500**	0.532	0.478
Max. dist.	0.750	**0.569**	0.835	0.844
Min. dist.	0.219	**0.439**	0.077	0.133

Table 6.4: Statistical results of the inter tests. The best result in each category is printed in bold.

Furthermore, it was examined if the inter score values can be improved by combining the tested perceptual image hash functions. Table 6.5 shows the statistical results. The score values of each image were summed up and divided by the number of the used perceptual image hash functions. It can be concluded that, regardless of which combination of hash functions is used, the Marr-Hildreth operator based image hash function delivers the best results.

	DCT + MH + Radial + BMB	DCT + Radial + BMB
Mean dist.	0.512	0.516
Max. dist.	0.656	0.701
Min. dist.	0.302	0.231

Table 6.5

6.3 Intra Score Distribution

Various common image operations were used to test the robustness of the perceptual image hash functions. JPEG compression and rotation are one of the most commonly used image operations which users employ to modify their images. They use these operations in order to reduce the file size of their images for instance. The rotation operation is especially often used in scientific papers to demonstrate the robustness of perceptual image hash functions. The horizontal flipping operation was used because it hardly changes the human perception of an image.

If the purpose of a benchmark is to measure the effects of an operation, it is important to keep in mind that the saving of an image using certain image formats (e.g. the JPEG image format) is an image modification or manipulation itself. Normally the process of applying an operation to an image is as follows. The image file is read from the hard disc. Then the image is decoded and stored in a custom raw format in the system memory. Subsequently, the operation (e.g. flipping the image horizontally) is applied. Finally, the image is converted from the custom raw format into a standardized image format and the result is written to the hard disc. It is important to use only image formats that do not use lossy compression for such benchmarks. An adequate image format would be e.g. PNG. Because image formats using lossy compression methods should not be used for such benchmarks the file size of the used image sets increases. For these benchmarks the used image sets were converted from the JPEG image format to the PNG image format. Therefore, the total file size of the chaos image set increased from 109.90 to 343.85MiB. As a result, the mean file size also increased from 2.44 to 7.64MiB. The total file size of the small chaos image set increased from 10.54 to 34.88MiB. Consequently the mean file size also increased from 3.51 to 11.63MiB. The chaos image set was used for all the score distribution charts. The small chaos image set was used for the effect of attack charts.

6.3.1 Horizontal Flipping

If an image is flipped, its binary representation is changed drastically, though its perception to the human visual system and its semantic meaning changes only minimally or not at all. Therefore, such an image operation is worth considering. Figure A.6 depicts the results of this benchmark. Table 6.6 summarizes them. None of the tested perceptual image hash functions is robust against horizontal flipping.

	DCT	MH	Radial	BMB
Run 0:				
Mean dist.	0.497	0.483	0.499	**0.315**
Max. dist.	**0.625**	0.658	0.732	0.703
Min. dist.	0.375	0.276	**0.042**	0.047

Table 6.6: Statistical results of the intra test. The images were changed by horizontally flipping them. The best result in each category is printed in bold.

6.3.2 Resizing

Figure A.7 shows the results of this benchmark and table 6.7 summarizes them. The images were changed by resizing the width to 1024 pixels. The height was adjusted proportionally. Bicubic interpolation was used. The radial variance based image hash function is not robust to resizing. This may stem from the fact that this hash function does not normalize the resolution of an image before extracting its features.

	DCT	MH	Radial	BMB
Run 0:				
Mean dist.	0.076	0.068	0.348	**0.012**
Max. dist.	0.219	0.271	0.670	**0.039**
Min. dist.	**0.000**	0.017	0.008	**0.000**

Table 6.7: Statistical results of the intra test. The images were changed by resizing the width to 1024 pixels. The height was adjusted proportionally. The best result in each category is printed in bold.

6.3.3 JPEG Compression

Figure A.8 depicts the results of this benchmark. Table 6.8 summarizes them. The images were changed using a JPEG quality setting of 80. Furthermore, the impact of the JPEG quality setting on the robustness of the perceptual image hash functions was investigated. Therefore the JPEG quality was gradually varied from 100 to 0. For each value of the quality parameter, the hash functions had to calculate the distance scores of the given images. The average distance scores as a function of the quality parameter are depicted in figure A.9. The radial variance based image hash function is almost not influenced at all by the quality parameter. Even when using a quality parameter of 0 the average distance score of this hash function is negligible. The same applies to the DCT and block mean value based image hash functions up to a quality parameter of 10. The Marr-Hildreth operator based image hash function performs the worst.

	DCT	MH	Radial	BMB
Run 0:				
Mean dist.	0.001	0.045	**0.000**	0.001
Max. dist.	0.031	0.253	**0.000**	0.008
Min. dist.	**0.000**	0.002	**0.000**	**0.000**

Table 6.8: Statistical results of the intra test. The images were changed using JPEG compression with a quality parameter of 80. The best result in each category is printed in bold.

6.3.4 Rotation

Figure A.10 depicts the results of this benchmark. Table 6.9 summarizes them. The images were rotated by 5° and bicubic interpolation was used. Using this kind of image operation none of the tested image hash functions is robust. Figure A.11 depicts the results when the angle of rotation is varied gradually. The block mean value based image hash function performs the best. When using a threshold of 0.3, it is robust against rotation up to 3°. The Marr-Hildreth operator based hash function is not robust at all. Even when rotating by only 1°, the average distance score is approximately 0.30.

	DCT	MH	Radial	BMB
Run 0:				
Mean dist.	0.335	0.486	0.358	**0.220**
Max. dist.	0.500	0.547	0.792	**0.375**
Min. dist.	0.062	0.337	**0.052**	0.117

Table 6.9: Statistical results of the intra test. The images were changed by rotating them by 5°. The best result in each category is printed in bold.

6.4 Summary

The newly implemented block mean based perceptual image hash function is faster than all the other functions. With regard to JPEG compression, rotation and resize operations it is either the most robust one or at least on a par with the other functions. None of the tested image hash functions is robust against flipping an image horizontally. Although the Marr-Hildreth operator based hash function is behind other functions when it comes to robustness, it has by far the most discriminative abilities.

The different properties of these hash functions can be leveraged by combining them. For instance, an image identification system could be implemented by combining the block mean value based image hash function together with the Marr-Hildreth operator based function. The block mean value based function would process the images in the first cycle. The candidates it identifies would then be passed on to the Marr-Hildreth operator based function. Such an image identification system would offer excellent performance in terms of speed and possess a great discriminative capability. Of course the Marr-Hildreth operator based function would be the limiting factor in terms of the robustness of the identification system. The inter score distribution can not be improved any further by combining the tested hash functions.

Chapter 7

Conclusion and Future Work

When a new perceptual hash function is published, adequate benchmarks and metrics are generally a part of the publication. As a matter of fact, benchmarks and metrics from different scientific papers can be hardly compared. There are numerous reasons. First of all, each publication uses different implementations of hash functions and attacks and different parameters for such. Secondly, the multimedia content (e.g. images) that is used differs. Finally, the scripts, programs, frameworks, operating systems and the hardware that is used to create the benchmarks and metrics are different. All of this shows that a ready-to-use benchmarking framework for perceptual hash functions is essential.

The contributions of this thesis are as follows. A benchmarking framework for perceptual image hash functions – called Rihamark – was proposed and implemented. Rihamark enables developers and decision makers to efficiently benchmark perceptual image hash functions and compare them. Rihamark was implemented in Java. Rihamark features a modular architecture. It can use analyzer, attack and algorithm plugins. For each of these plugin classes, numerous plugins were implemented. A block mean value based perceptual image hash function, which was previously proposed in [44], was implemented in C/C++. The implementation was integrated into pHash. Nonetheless, the implementation is self-contained to a large extent and could be compiled and distributed without pHash if required. The JNI interface of pHash was modified and extended in order to be able to write a Rihamark plugin for it.[1] Finally, four different perceptual image hash functions were benchmarked using Rihamark and the results were discussed. A DCT based hash function, a Marr-Hildreth operator based hash function, a radial variance based image hash function and the newly implemented block mean value based image hash function were used.

The newly implemented block mean based perceptual image hash func-

[1] As of the date of the writing of this thesis, these changes have not been integrated into the upstream version of pHash.

tion is faster than all the other functions. With regard to JPEG compression, rotation and resize operations it is either the most robust one or at least on par a with the other functions. None of the tested image hash functions is robust against flipping an image horizontally. Although the Marr-Hildreth operator based hash function is behind other functions when it comes to robustness, it has by far the most discriminative abilities.

The different properties of these hash functions can be leveraged by combining them. For instance, an image identification system could be implemented by combining the block mean value based image hash function together with the Marr-Hildreth operator based function. The block mean value based function would process the images in the first cycle. The candidates it identifies would then be passed on to the Marr-Hildreth operator based function. Such an image identification system would offer excellent performance in terms of speed and possess a great discriminative capability. Of course the Marr-Hildreth operator based function would be the limiting factor in terms of the robustness of the identification system. The inter score distribution can not be improved any further by combining the tested hash functions.

Future work could improve the usability of Rihamark by implementing a test plan and report generator. At the moment, the user has to create the test plan himself, depending on the benchmark he wants to create (e.g. a speed or an effect of attack benchmark). So the user has to know the required composition of a test plan. A test plan generator could automate this process. The user then would only need to specify that he wants a test plan to be created that is appropriate for a given benchmark setting. That is e.g., an effect of attack benchmark for the rotation attack, whereby the angle is gradually varied from 0 to 360 degrees using steps of 10 degrees. Moreover, the radial variance and the block mean value based image hash function should be used. A report generator would further improve the usability of Rihamark. A report would consist of several benchmarks (test plans). If a user wants to create a new report, he would only need to choose which algorithms should be evaluated using which benchmarks. The output of Rihamark then should be e.g. a Hypertext Markup Language (HTML) or PDF formated-document outlining all the results including charts and tables. Another possible future direction is to make Rihamark applicable to perceptual hash functions for other multimedia content such as audio or video. pHash is written in C/C++. Although it supports Unix and Microsoft Windows operating systems it has to be compiled separately for each target platform. Therefore, one could implement some hash functions of pHash in Java and compare the implementations in terms of speed.

Appendix A

Charts of the Benchmark Results

A.1 Speed

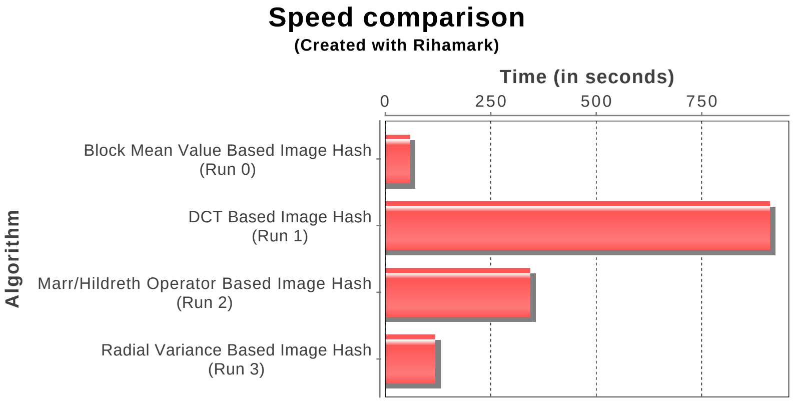


Figure A.1: Results of the speed benchmark.

A.2 Inter Score Distribution

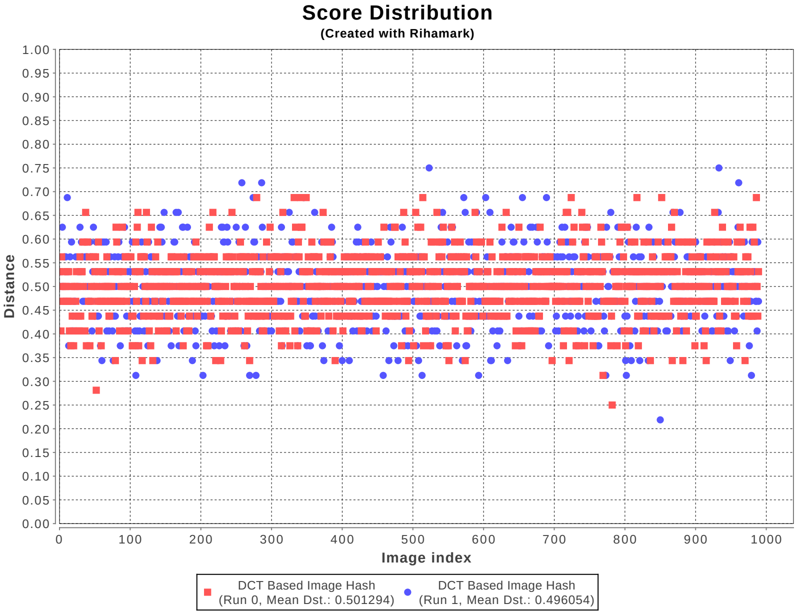


Figure A.2: Results of the DCT based image hash function for two inter tests (the chaos and the duck image sets were used).

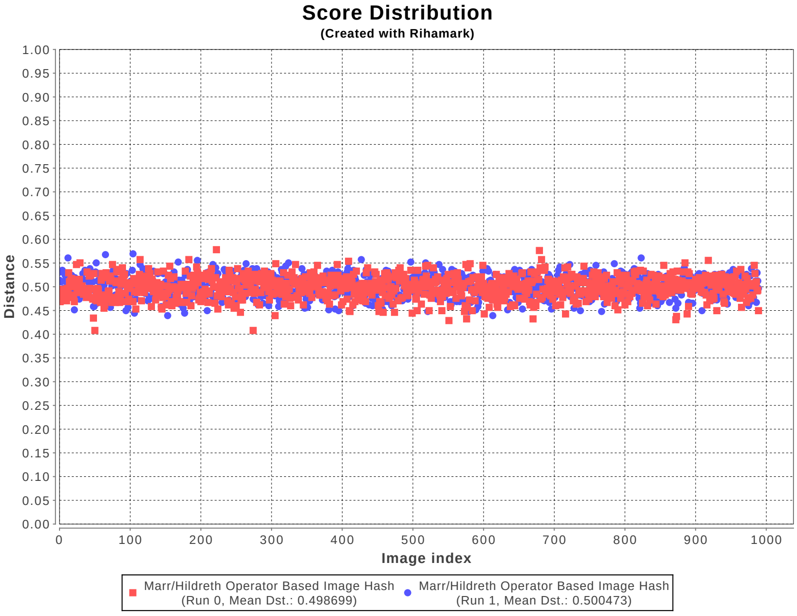


Figure A.3: Results of the Marr-Hildreth operator based image hash function for two inter tests (the chaos and the duck image sets were used).

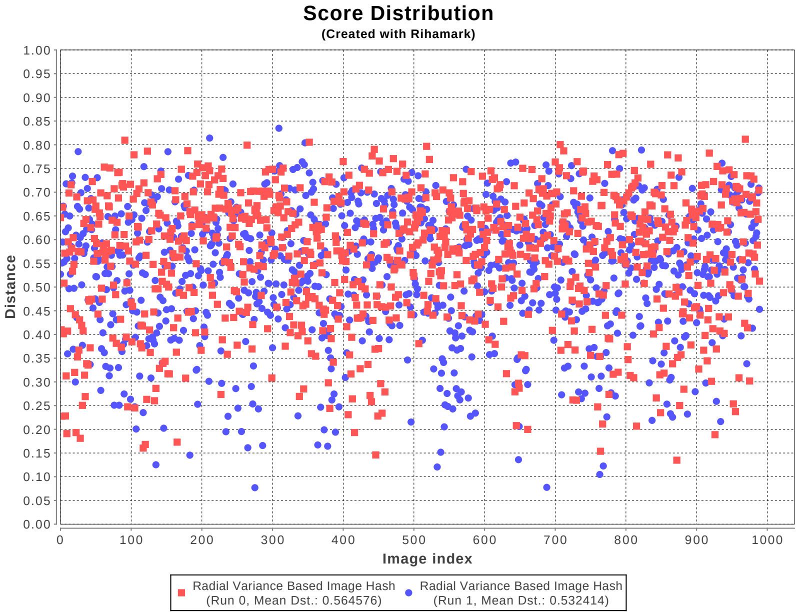


Figure A.4: Results of the radial variance based image hash function for two inter tests (the chaos and the duck image sets were used).

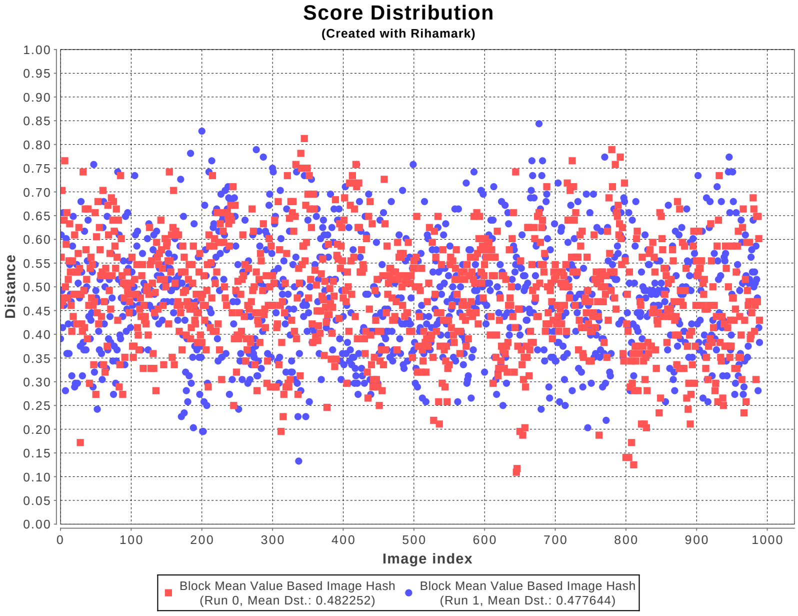


Figure A.5: Results of the block mean value based image hash function for two inter tests (the chaos and the duck image sets were used).

A.3 Intra Score Distribution

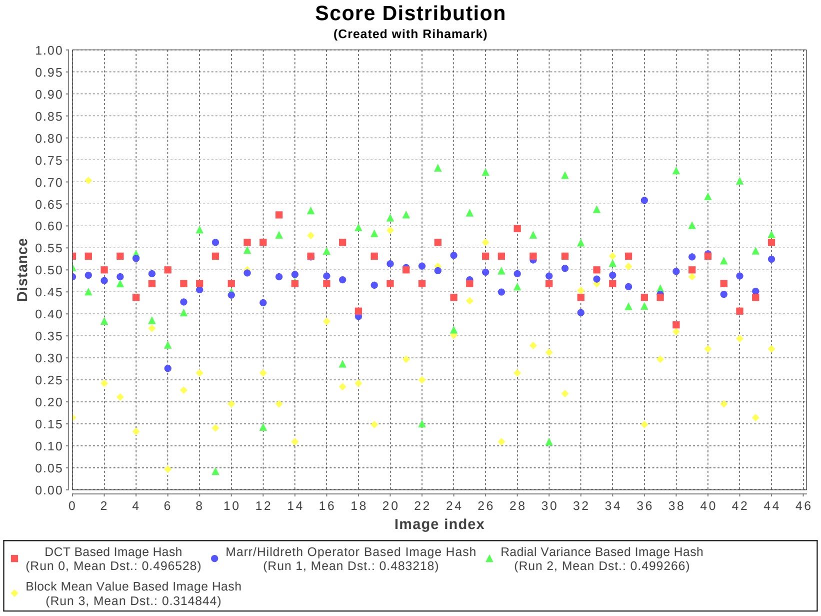


Figure A.6: The images were changed by horizontally flipping them.

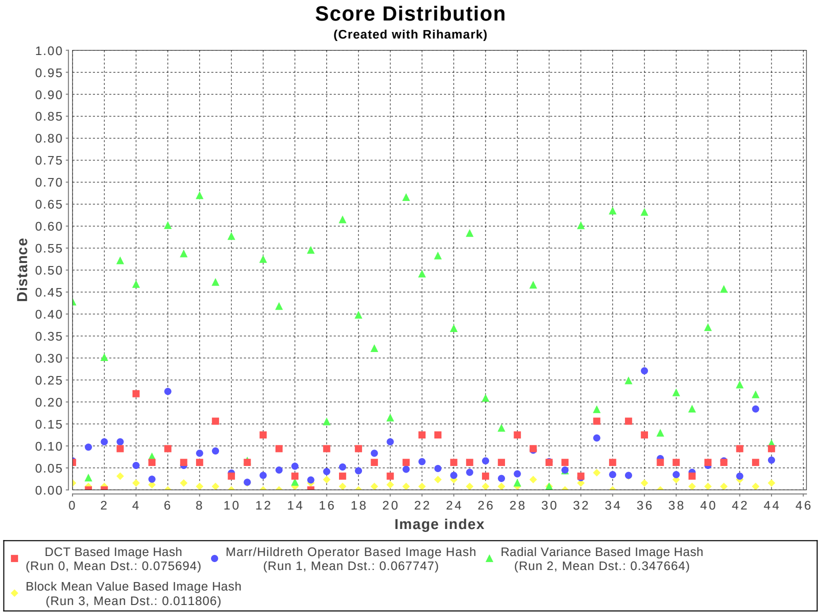


Figure A.7: The width of the images was resized to 1024 pixels. The height was adjusted proportionally.

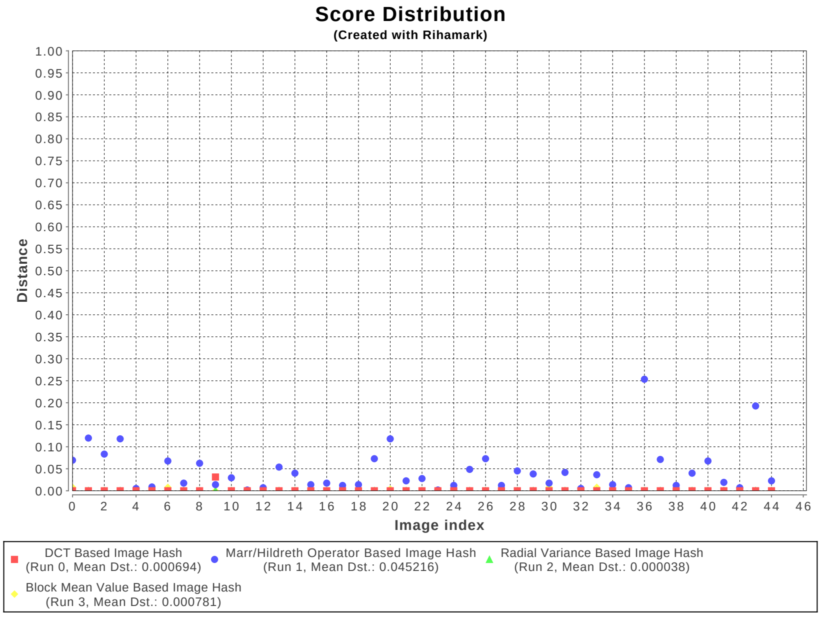


Figure A.8: The images were changed using JPEG compression with a quality parameter of 80.

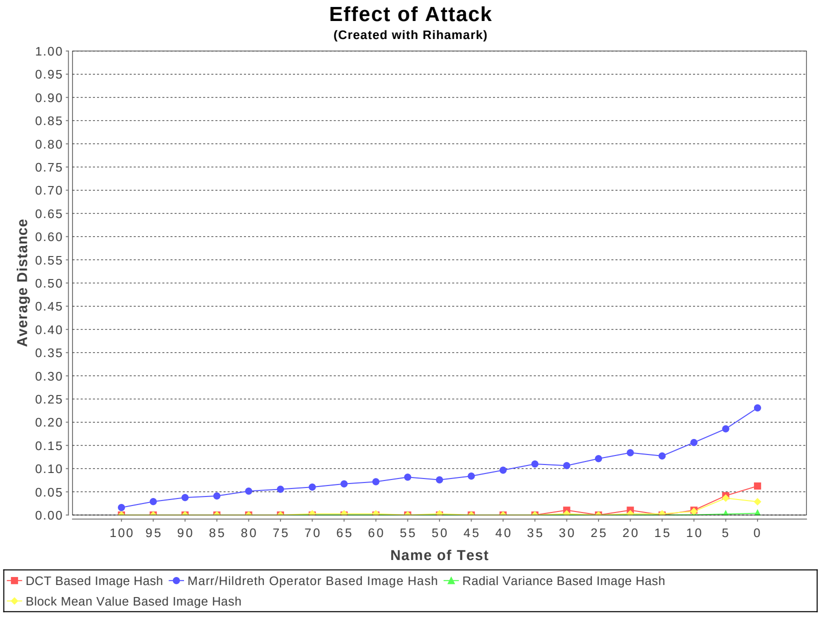


Figure A.9: The JPEG quality parameter was gradually varied from 100 to 0.

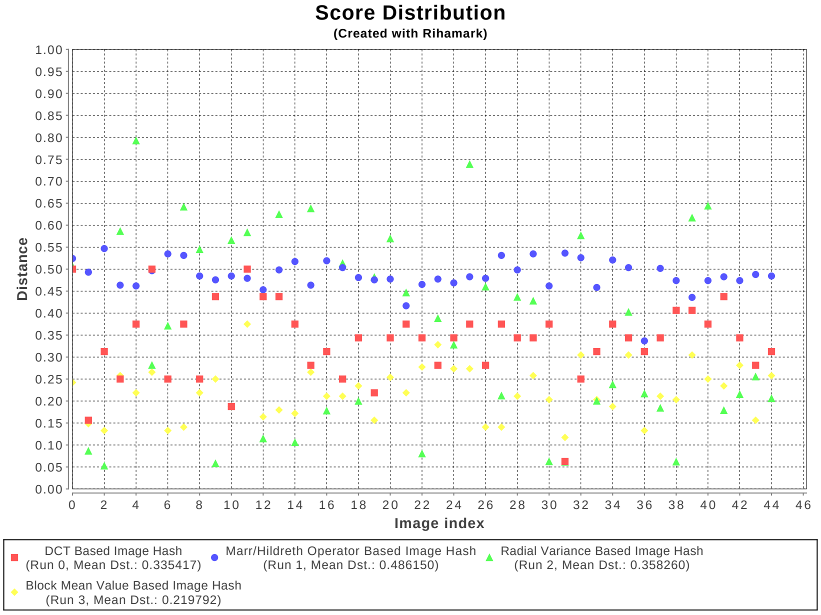


Figure A.10: The images were rotated by 5 degrees.

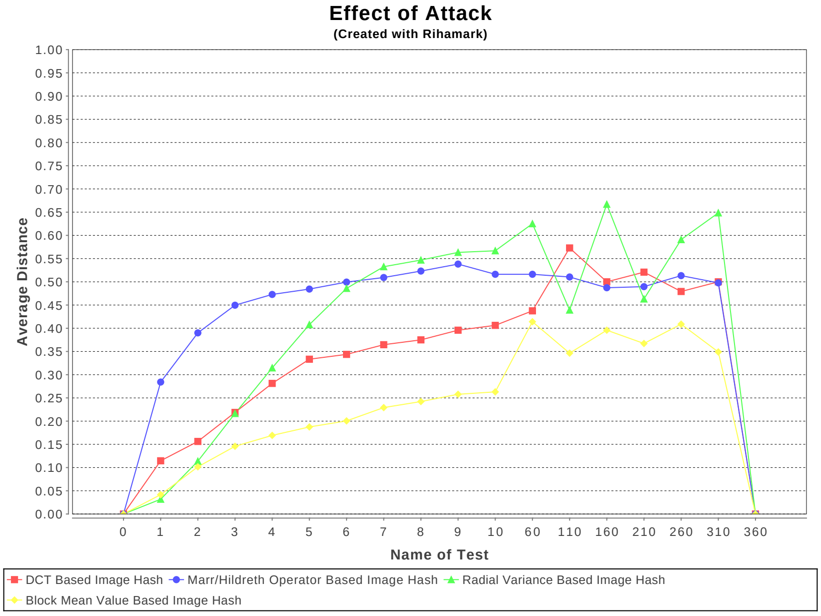


Figure A.11: The angle was gradually varied $(0°, 1°, \ldots, 10°, 60°, \ldots, 360°)$.

Appendix B

Listings

```
int ph_dct_imagehash
(
  const char* file,
  ulong64 &hash
);

int ph_hamming_distance
(
  const ulong64 hash1,
  const ulong64 hash2
);

uint8_t* ph_mh_imagehash
(
  const char *filename,
  int &N,
  float alpha = 2.0f,
  float lvl   = 1.0f
);

double ph_hammingdistance2
(
  uint8_t *hashA,
  int lenA,
  uint8_t *hashB,
  int lenB
);

int ph_image_digest
(
  const char *file,
  double sigma,
  double gamma,
  Digest &digest,
  int N=180
);

```

```
typedef struct ph_digest {
    char *id;
    uint8_t *coeffs;
    int size;
} Digest;

int ph_dct
(
  const Features &fv,
  Digest &digest
);

int ph_crosscorr
(
  const Digest &x,
  const Digest &y,
  double &pcc,
  double threshold = 0.90
);

PHRetCode bmb_imagehash
(
  const char *const file,
  bmb_hashopts *hashopts,
  uint8_t method,
  BinHash **ret_hash
);
```

Listing B.1: Important declarations in *pHash.h*.

```
package org.pHash;
import java.io.*;

public class pHash {

  /*** begin JNI bindings ***/

  // library management
  private native static void pHashInit();
  private native static void cleanup();

  // image hash distance (can be used for all image hashes)
  private native static double imageDistance(ImageHash hash1,
      ⟩ImageHash hash2);

  // DCT image hashing
  private native static DCTImageHash    dctImageHash(String
      ⟩file);

  <...>

  /*** end JNI bindings ***/

  static {
    System.loadLibrary("pHash-jni");
    pHashInit();
  }

  <...>

  // DCT image hash

  public static DCTImageHash getDCTImageHash(File file) {
    DCTImageHash imHash = dctImageHash(file.toString());
    return imHash;
  }

  public static double getDCTImageHashDistance(DCTImageHash
      ⟩hashA, DCTImageHash hashB) {
    return(imageDistance(hashA, hashB));
  }

  public static double getDCTImageHashDistance(String file1,
      ⟩String file2) {
    DCTImageHash imHash1 = dctImageHash(file1);
    DCTImageHash imHash2 = dctImageHash(file2);
    return(imageDistance(imHash1,imHash2));
  }

  <...>

} // end of class
```

Listing B.2: Java API of pHash (part of file *pHash.java)*.

```
package rmk.core;

/**
 * The Core uses this class to signal the user
 * interface various events.
 *
 * @author Christoph Zauner
 */
public interface TaskReport {

  /**
   * Used to signal that the task has finished.
   *
   * @param finishedSuccessfully
   *        {@code true} if the task has finished
   *        successfully; otherwise {@code false}
   */
  public void reportFinished(boolean finishedSuccessfully);

  /**
   * Used to report the progress in terms of
   * completed jobs.
   *
   * @param progress
   *        the progress the task is now at
   */
  public void reportProgress(int progress);

} // end of class
```

Listing B.3: The `TaskReport` interface. Every user interface of the Rihamark Core has to implement this interface.

```
  public Rotation() {
    super(name, bundleName, description);
    Debug.debugPrint(LogLevel.DEBUG2, "Constructor in " + this.
        ⟩getClass().getCanonicalName());

    if(!deserialization) {
      // argument
      angle = new SpinnerInt(45, "Degrees", 0, 360, 1);
      addSPArgument(angle);

      // argument
      interpolation = new ComboBox("None", "Interpolation", new
          ⟩ String[] {"None", "Bilinear", "Bicubic"});
      addSPArgument(interpolation);

      // argument
      enlarge = new CheckBox(true, "Enlarge Image to Fit Result
          ⟩");
      addSPArgument(enlarge);
    }
  }
```

Listing B.4: Constructor of the `Attack` service provider `Rotation`.

Appendix C

CD-ROM Content

C.1 Miscellaneous

Path: ./

thesis.pdf . . . . Master's thesis (this document in PDF format).

thesis_colour.pdf . . . . This is a more colourful version of the master's thesis. More of its figures are coloured.

Path: benchmarks/

./ . . . . Contains the test plans that were used for benchmarking and the detailed results of the benchmarks.

Path: image_sets/

./ . . . . Contains the image sets that were used for benchmarking.

Path: litarchive/

./ . . . . Contains the archived literature.

C.2 pHash

Path: phash/

./ . . . . This directory contains the modified source code of pHash. It is based on upstream version 0.9. pHash was modified in order to be used together with the Rihamark benchmarking framework. Additionally a new perceptual image hash function was implemented.[1]

C.3 Rihamark

Path: rihamark_dev/

devdoc/ . . . . Unified Modelling Language (UML) class diagrams of the Rihamark benchmarking framework.

rmk-core/ . . . . Netbeans project folder of the `rmk.core` package. This package implements the core of the Rihamark benchmarking framework. Additionally the `SPArguments`, the `SPInterfaces` and the `SPMenu` Java packages are part of this project folder.

rmk-gui/ . . . . Netbeans project folder of the `rmk.gui` package. This package implements a GUI for the Rihamark benchmarking framework.

rmk-plugins/ . . . . . . Netbeans project folder of the `rmk.plugins` package. This package implements the default plugins of the Rihamark benchmarking framework.

Path: rihamark_bundle/

./ . . . . Contains the binary distribution of the Rihamark benchmarking framework.

[1] As of the date of the writing of this thesis, these changes have not been integrated into the upstream version of pHash.

Appendix D

Remarks Concerning the Notation

Table D.1 – D.3 summarize uncommon and potential ambiguous symbols used in mathematics and computer science.

Symbol	**Name**	**Read as**	**Field**
	Explanation		
	Example		
$\gg$, $\ll$	very strict inequality	is much less than, is much greater than	order theory
	$x \gg y$ means x is much greater than y. $x \ll y$ means x is much less than y.		
	$0.005 \ll 100000$.		
∇f $(x_1,$ $\ldots$ $, x_n)$	gradient	del, nabla, gradient of	vector calculus
	$\nabla f(x_1, \ldots, x_n)$ is the vector of partial derivatives $(\partial f/\partial x_1, \ldots, \partial f/\partial x_n)$.		
	If $f(x, y, z) := 3xy + z^2$, then $\nabla f = (3y, 3x, 2z)$.		

Table D.1: Remarks concerning the notation (part 1).

<table>
<tr><th rowspan="3">Symbol</th><th>Name</th><th>Read as</th><th>Field</th></tr>
<tr><th colspan="3">Explanation</th></tr>
<tr><th colspan="3">Example</th></tr>
<tr><td rowspan="3">$\vert$</td><td>conditional probability</td><td>given</td><td>probability</td></tr>
<tr><td colspan="3">$P(a|b)$ means the probability of the event a occurring given that b occurs.</td></tr>
<tr><td colspan="3">If x is a uniformly random day of the year, then $P(x \text{ is May 25 } |x \text{ is in May }) = \frac{1}{31}$.</td></tr>
<tr><td rowspan="3">$\forall$</td><td>universal quantification</td><td>for all</td><td>predicate logic</td></tr>
<tr><td colspan="3">$\forall x : P(x)$ means $P(x)$ is true for all x.</td></tr>
<tr><td colspan="3">$\forall n \in \mathbb{N} : n^2 \geq n$.</td></tr>
<tr><td rowspan="3">$\prod$</td><td>product</td><td>product over … from … to … of</td><td>arithmetic</td></tr>
<tr><td colspan="3">$\prod_{k=1}^{n} a_k$ means $a_1 * a_2 * \cdots * a_n$.</td></tr>
<tr><td colspan="3">$\prod_{k=1}^{3} = 1 * 2 * 3 = 6$.</td></tr>
<tr><td rowspan="3">$\binom{n}{k}$</td><td>binomial coefficient</td><td>n choose k</td><td>combinatorics</td></tr>
<tr><td colspan="3">The number of k-element subsets that can be drawn from a set with n-elements. Thereby the sequence is irrelevant and the drawn elements are not put back. $\binom{n}{k}$ is defined as $\frac{n!}{k!(n-k)!}, \forall n \geq k$, where $k, n \in \mathbb{N}$.</td></tr>
<tr><td colspan="3">$\binom{4}{2} = 6$.</td></tr>
</table>

Table D.2: Remarks concerning the notation (part 2).

<table>
<tr><th rowspan="3">Symbol</th><th>Name</th><th>Read as</th><th>Field</th></tr>
<tr><th colspan="3">Explanation</th></tr>
<tr><th colspan="3">Example</th></tr>
<tr><td rowspan="3">$\partial f/\partial x_i$</td><td>partial derivative</td><td>partial, d</td><td>calculus</td></tr>
<tr><td colspan="3">$\partial f/\partial x_i$ means the partial derivative of f with respect to x_i, where f is a function on $(x1, \ldots, x_n)$.</td></tr>
<tr><td colspan="3">If $f(x, y) := x^2y$, then $\partial f/\partial x = 2xy$.</td></tr>
<tr><td rowspan="3">#,
$|\ldots|$</td><td>cardinality</td><td>cardinality of; size of; order of</td><td>set theory</td></tr>
<tr><td colspan="3">$\#X$ (or $|X|$) means the cardinality of the set X.</td></tr>
<tr><td colspan="3">$\#\{3, 5, 7, 9\} = 4$.</td></tr>
<tr><td rowspan="3">!</td><td>factorial</td><td>factorial</td><td>combinatorics</td></tr>
<tr><td colspan="3">$n! := 1 * 2 * \cdots * n$.</td></tr>
<tr><td colspan="3">$3! = 6$.</td></tr>
<tr><td rowspan="3">$*$</td><td>convolution</td><td>convolution, convolved with</td><td>functional analysis</td></tr>
<tr><td colspan="3">$f * g$ means the convolution of f and g.</td></tr>
<tr><td colspan="3">$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$.</td></tr>
</table>

Table D.3: Remarks concerning the notation (part 3).

Acronyms

API

__A__pplication __P__rogramming __I__nterface. Page 29, 33, 85

BER

__B__it __E__rror __R__ate... For a definition see section 4.1. Page 14, 15, 35

CBIR

__C__ontent-__b__ased __i__mage __r__etrieval... Process of retrieving desired digital images from a large collection on the basis of syntactical image features. Such features can be colours, shapes, textures, or any other information that can be derived from the image itself. Page 14

DCT

__d__iscrete __C__osine __t__ransform. Page viii, xii, xiii, 12, 21, 22, 27, 29–31, 33, 61, 62, 65, 67, 70

DET

__D__etection __E__rror __T__rade-off... For a definition see section 4.1. Page 38

DFT

__d__iscrete __F__ourier __T__ransform. Page 21

DOG

__d__ifference __o__f __G__aussian... For a definition see section 3.1.2. Page 25, 26

DWT

__d__iscrete __w__avelet __t__ransform... Any wavelet transform for which the wavelets are discretely sampled. Page 29

EER

__E__qual __E__rror __R__ate... For a definition see section 4.1. Page 40

EP

__E__quality __P__ercentage... For a definition see section 4.1. Page 16

Exif

Ex*changeable* ***i****mage* ***f****ile format*... Specification for image file formats used by digital cameras. It adds support for specific metadata tags. Page 8

FAR

F*alse* ***A****ccept* ***R****ate*... For a definition see section 4.1. Page viii, 34–39, 41

FRR

F*alse* ***R****eject* ***R****ate*... For a definition see section 4.1. Page viii, 35–39, 41, 43, 44

GNU

G*NU is* ***n****ot* ***U****nix*. Page x, 29, 33

GNU AGPL

GNU ***A****ffero* ***G****eneral* ***P****ublic* ***L****icense*. Page 85

GNU GPL

GNU ***G****eneral* ***P****ublic* ***L****icense*. Page 86

GNU LGPL

GNU ***L****esser* ***G****eneral* ***P****ublic* ***L****icense*. Page 85

GUI

G*raphical* ***U****ser* ***I****nterface*. Page 47, 48, 52, 53, 55, 57, 80

HTML

H*yper****t****ext* ***M****arkup* ***L****anguage*. Page 68

JNI

J*ava* ***N****ative* ***I****nterface*. Page x, 33, 47, 67

JPEG

J*oint* ***P****hotographic* ***E****xperts* ***G****roup*... Colloquial term for an international standard for image compression. A file format to save the resulting data is not part of this standard. Page ix, xi, 6, 7, 22, 45, 50, 51, 56, 63–66, 68, 73

LoG

L*aplacian* ***o****f* ***G****aussian*... For a definition see section 3.1.2. Page 24–26, 31

MAC

M*essage* ***A****uthentication* ***C****ode*. Page 4

MDC
***M**odification **D**etection **C**ode*... See [27, p. 323]. Page 17

MiB
***M**e**bi**byte*... The term Mebibyte is an abbreviation for **me**ga **bi**nary **byte**. 1 Mebibyte = 2^{20} Bytes. Page 60, 61, 63

PCC
***P**eak of **C**ross **C**orrelation*... For a definition see section 4.1. Page 14, 17, 32

PDF
***P**ortable **D**ocument **F**ormat*... Widely used electronic document format. It has the ability to reproduce high quality output on a variety of different platforms. Page 57, 68, 79, 85

PHABS
***P**erceptual **H**ashing **A**lgorithms **B**enchmark **S**uite*. Page 46

PNG
***P**ortable **N**etwork **G**raphics*. Page 51, 56, 57, 63

ROC
***R**eceiver **O**perating **C**haracteristic*... For a definition see section 4.1. Page viii, 36–40, 57, 58

SPI
***S**ervice **P**rovider **I**nterface*... For a definition see section 5.2. Page 53

SVD
***s**ingular **v**alue **d**ecomposition*... A factorization of a rectangular real or complex matrix. Page 29

SVG
***S**calable **V**ector **G**raphics*... A standard language for describing two-dimensional graphics in XML format. It is a recommendation of the World Wide Web Consortium. Page 57, 85

UML
***U**nified **M**odeling **L**anguage*. Page viii, 49, 54, 80

XML
*E**x**tensible **M**arkup **L**anguage*. Page 50, 52

Glossary

ID3

A metadata container that is most often used in conjunction with the MP3 audio file format. It allows information such as the title, artist, album, track number, and other information about the file to be stored in the file itself. Page 8

Programs

Batik
: `http://xmlgraphics.apache.org/batik/`; A Java library that provides an API to generate, modify and display SVG files; Version: 1.7 (10. Jan. 2008); Licence: Apache Licence 2.0. Page 57

CImg
: `http://cimg.sourceforge.net/`; C++ library for image processing; Version: 1.3.4 (8. Apr. 2010); Licence: CeCILL-C or CeCILL (both are open source); Operating System: Linux, Microsoft Windows, Mac OS X. Page 30, 31

Gnuplot
: `http://www.gnuplot.info/`; A command-line driven graphing utility; Licence: own licence (open source). Page 52

Image Hashing Toolbox
: `http://users.ece.utexas.edu/~bevans/`; A Matlab demo program for various perceptual image hashing functions; Version: 0.1 beta (18. Juni 2006); Dependencies: MATLAB 6.1 or 6.5 or higher, Image Processing Toolbox, Wavelet Toolbox. Page 29

ImageJ
: `http://rsb.info.nih.gov/ij/`; A Java-based image processing program; Version: 1.43u (24. Apr. 2010); Licence: Public Domain. Page 56

iText
: `http://itextpdf.com/`; A Java library that provides an API to generate and modify e.g. PDF files; Version: 5.0.2 (13. Apr. 2010); Licence: GNU GNU Affero General Public Licence (GNU AGPL) (terms of use: `http://itextpdf.com/terms-of-use/index.php`). Page 57

JFreeChart
: `http://www.jfree.org/jfreechart/`; A Java library to create complex charts; Version: 1.0.13 (20. Apr. 2009); Licence: GNU Lesser General Public Licence (GNU LGPL). Page 57

Matlab

`http://www.mathworks.com/products/matlab/`; A computing environment and programming language to solve mathematical problems and visualize the results; Version: 7.10 (5. Mar. 2010); Licence: proprietary; Operating Systems: Linux, Microsoft Windows, Mac OS X. Page 29, 47, 85

pHash

`http://www.phash.org`; A C/C++ library implementing various perceptual image hash functions; Version: 0.9.0 (28. Mar. 2010); Licence: GNU General Public Licence (GNU GPL); Operating System: Linux, Microsoft Windows, Mac OS X. Page x, xii, xiii, 29–33, 56, 59, 67, 68, 79, 80

R

`http://www.r-project.org/`; A software environment for statistical computing and graphics; Licence: GNU GPL. Page 52

Simple

`http://simple.sourceforge.net`; Simple is an XML serialization and configuration framework for Java; Version: 2.3.3 (5. Mar. 2010); Licence: Apache Licence 2.0. Page 52

Bibliography

[1] Bhattacharjee, S. and Kutter, M.: *Compression tolerant image authentication.* In *Proceedings of the International Conference on Image Processing (ICIP)*, vol. 1, pp. 435–439. IEEE, Oct. 1998.

[2] Bolle, R., Connell, J., Pankanti, S., Ratha, N., and Senior, A.: *Guide to Biometrics.* Springer, 2004, ISBN 0387400893.

[3] Bourke, P.: *Cross Correlation*, Aug. 1996. `http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/correlate/`, accessed on 31. May 2010, copy on CD-ROM (lit-003).

[4] Bovik, A. (ed.): *The Essential Guide to Image Processing.* Academic Press, 2009.

[5] Caldelli, R., Vogel, T., Dittmann, J., Thiemert, S., Solachidis, V., Voloshynovskiy, S., Deguillaume, F., Pun, T., Minguillon, J., Megias, D., Schmucker, M., and Steinebach, M.: *First summary report on authentication.* Tech. Rep. D.WVL.6, ECRYPT, Jan. 2005.

[6] Cano, P.: *Content-Based Audio Search: from Fingerprinting to Semantic Audio Retrieval.* PhD thesis, Universitat Pompeu Fabra, 2007. `http://mtg.upf.edu/files/publications/34ac8d-PhD-Cano-Pedro-2007.pdf`.

[7] Cano, P., Kaltenbrunner, M., Gouyon, F., and Batlle, E.: *On the use of fastmap for audio information retrieval and browsing.* In *Proceedings of the International Conference on Music Information Retrieval (ISMIR)*, Oct. 2002. `http://mtg.upf.edu/files/publications/ismir02-pcano.pdf`.

[8] Coskun, B. and Memon, N.: *Confusion/diffusion capabilities of some robust hash functions.* In *Proceedings of the Conference on Information Sciences and Systems (CISS)*, pp. 1188–1193. IEEE, Mar. 2006.

[9] Coskun, B. and Sankur, B.: *Robust video hash extraction.* In *Proceedings of the Signal Processing and Communications Applications Conference*, pp. 292–295. IEEE, Apr. 2004.

[10] Cox, I.J., Doërr, G.J., and Furon, T.: *Watermarking is not cryptography.* In Shi, Y.Q. and Jeon, B. (eds.): *Proceedings of the International Workshop on Digital Watermarking (IWDW)*, vol. 4283 of *Lecture Notes in Computer Science*, pp. 1–15. Springer, Nov. 2006, ISBN 3-540-48825-1.

[11] Cox, I.J., Miller, M.L., and Bloom, J.A.: *Digital Watermarking.* Morgan Kaufmann, 2002, ISBN 1558607145.

[12] Daugman, J.G. and Williams, G.O.: *A proposed standard for biometric decidability.* In *CardTech SecurTech (Atlanta, GA)*, pp. 223–234, May 13-16, 1996.

[13] Drakos, N. and Moore, R.: *Definition of DCT*, n.Y. `http://fourier.eng.hmc.edu/e161/lectures/dct/node1.html`, accessed on 26. May 2010, copy on CD-ROM (lit-002).

[14] Fridrich, J.: *Robust bit extraction from images.* In *Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS)*, vol. 2, pp. 536–540. IEEE, June 1999.

[15] Fridrich, J. and Goljan, M.: *Robust hash functions for digital watermarking.* In *Proceedings of the International Conference on Information Technology (ITCC)*, pp. 178–183. IEEE, Mar. 2000, ISBN 0-7695-0540-6.

[16] Friedman, G.: *The trustworthy digital camera: restoring credibility to the photographic image.* IEEE Transactions on Consumer Electronics, 39(4):905–910, Nov. 1993, ISSN 0098-3063.

[17] Haitsma, J. and Kalker, T.: *A highly robust audio fingerprinting system.* In *Proceedings of the International Conference on Music Information Retrieval (ISMIR).* The International Society for Music Information Retrieval, Oct. 2002. `http://ismir2002.ismir.net/proceedings/02-FP04-2.pdf`.

[18] Haitsma, J., Kalker, T., and Oostveen, J.: *An efficient database search strategy for audio fingerprinting.* In *Proceedings of the Workshop on Multimedia Signal Processing (MMSP)*, pp. 178–181. IEEE, Dec. 2002.

[19] Hamming, R.W.: *Error detecting and error correcting codes.* The Bell System Technical Journal, XXIX(2), Apr. 1950. `http://guest.engelschall.com/~sb/hamming/`.

[20] Kilburn, D.: *Dirty linen, dark secrets.* Adweek, 38(40):35–40, Oct. 1996.

[21] Klinger, E. and Starkweather, D.: *pHash.org: Development Guide.* Aetilius, Inc., n.Y. `http://phash.org/docs/howto.html`, accessed on 15. Apr. 2010, copy on CD-ROM (lit-000).

[22] Lefèbvre, F., Macq, B., and Legat, J.D.: *RASh: RAdon Soft Hash algorithm.* In *Proceedings of the European Signal Processing Conference (EUSIPCO)*, vol. I, pp. 299–302. European Association for Signal Processing, Sept. 2002.

[23] Lewis, J.P.: *Fast template matching.* Vision Interface, pp. 120–123, 1995.

[24] Lin, C.Y. and Chang, S.F.: *A robust image authentication method distinguishing JPEG compression from malicious manipulation.* IEEE Transactions on Circuits and Systems for Video Technology, 11(2):153–168, Feb. 2001, ISSN 1051-8215.

[25] Macq, B. and Quisquater, J.J.: *Cryptology for digital TV broadcasting.* Proceedings of the IEEE, 83(6):944–957, June 1995, ISSN 0018-9219.

[26] Meixner, A. and Uhl, A.: *Robustness and security of a wavelet-based CBIR hashing algorithm.* In *Proceedings of the Workshop on Multimedia and Security (MM&SEC)*, pp. 140–145. Association for Computing Machinery, Sept. 2006.

[27] Menezes, A.J., Vanstone, S.A., and Oorschot, P.C.V.: *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996, ISBN 0849385237.

[28] Mihçak, M. and Venkatesan, R.: *New iterative geometric methods for robust perceptual image hashing.* In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, vol. 2200 of *Lecture Notes in Computer Science*, pp. 13–21. Springer, 2001.

[29] Mihçak, M.K. and Venkatesan, R.: *A perceptual audio hashing algorithm: A tool for robust audio identification and information hiding.* In Moskowitz, I.S. (ed.): *Proceedings of the 4th International Information Hiding Workshop (IHW)*, vol. 2137 of *Lecture Notes in Computer Science*, pp. 51–65. Springer, 2001, ISBN 3-540-42733-3.

[30] Miller, M., Rodriguez, M., and Cox, I.: *Audio fingerprinting: nearest neighbor search in high dimensional binary spaces.* In *Proceedings of 2002 IEEE Workshop on Multimedia Signal Processing (MMSP)*, pp. 182–185. IEEE, Dec. 2002.

[31] Min, Z., Changjia, C., and Jinkang, J.: *Fake servers in EDonkey networks.* In *Proceedings of the 5th International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, pp. 1–7. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, ISBN 978-963-9799-26-4.

[32] Monga, V.: *Perceptually Based Methods for Robust Image Hashing.* PhD thesis, University of Texas, Aug. 2005.

[33] Nickel, C.: *Authentifizierung von Bildern mit Fingerprinting-Verfahren.* Master's thesis, Technische Universität Darmstadt, 2007.

[34] Radon, J.: *On the determination of functions from their integral values along certain manifolds.* IEEE Transactions on Medical Imaging, 5(4):170–176, Dec. 1986, ISSN 0278-0062.

[35] Roover, C.D., Vleeschouwer, C.D., Lefèbvre, F., and Macq, B.M.: *Robust image hashing based on radial variance of pixels.* In *Proceedings of the International Conference on Image Processing (ICIP)*, vol. 3, pp. 77–80. IEEE, Sept. 2005.

[36] Roy, S. and Sun, Q.: *Robust hash for detecting and localizing image tampering.* In *Proceedings of the International Conference on Image Processing (ICIP)*, pp. 117–120. IEEE, 2007.

[37] Schneider, M. and Chang, S.F.: *A robust content based digital signature for image authentication.* In *Proceedings of the International Conference on Image Processing (ICIP)*, vol. 3, pp. 227–230. IEEE, Sept. 1996.

[38] Standaert, F.X., Lefèbvre, F., Rouvroy, G., Macq, B.M., Quisquater, J.J., and Legat, J.D.: *Practical evaluation of a radial soft hash algorithm.* In *Proceedings of the International Symposium on Information Technology: Coding and Computing (ITCC)*, vol. 2, pp. 89–94. IEEE, Apr. 2005.

[39] Steinebach, M., Zmudzinski, S., and Neichtadt, S.: *Robust-audio-hash synchronized audio watermarking.* In Fernández-Medina, E. and Valle, M.I.Y. del (eds.): *Proceedings of the 4th International Workshop on Security in Information Systems (WOSIS)*, pp. 58–66. INSTICC Press, May 2006, ISBN 978-972-8865-52-8.

[40] Swaminathan, A., Mao, Y., and Wu, M.: *Robust and secure image hashing.* IEEE Transactions on Information Forensics and Security, 1(2):215–230, 2006.

[41] Uhl, A. and Zhang, H.: *Progress of forensic tracking techniques.* Tech. Rep. D.WVL.17, ECRYPT, Feb. 2007.

[42] Venkatesan, R., Koon, S.M., Jakubowski, M.H., and Moulin, P.: *Robust image hashing.* In *Proceedings of the International Conference on Image Processing (ICIP)*, vol. 3, pp. 664–666. IEEE, Sept. 2000.

[43] Wayman, J., Jain, A., Maltoni, D., and Maio, D.: *Biometric Systems – Technology, Design and Performance Evaluation*. Springer, 2005, ISBN 1852335963.

[44] Yang, B., Gu, F., and Niu, X.: *Block mean value based image perceptual hashing*. In *Proceedings of the International Conference on Intelligent Information Hiding and Multimedia Multimedia Signal Processing (IIH-MSP)*, pp. 167–172. IEEE, 2006, ISBN 0-7695-2745-0.

[45] Zhang, H., Schmucker, M., and Niu, X.: *The design and application of phabs: A novel benchmark platform for perceptual hashing algorithms*. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*, pp. 887–890. IEEE, July 2007.