TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Engineering

IAG70LT

Viktor Popkov 132458IAPM

POSSIBLE APPLICATION OF PERCEPTUAL IMAGE HASHING

Master thesis

Supervisor: Vladimir Viies

Associate Professor

Tallinn 2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

_____ _____

(kuupäev) (allkiri)

Abstract

The aim of this work is to discuss the theoretical basis of perceptual image hashing by proposing the image hashing framework and researching existing perceptual image hash functions. The research will show the main differences between perceptual and cryptographic hash functions and their applications scenarios. Moreover, we will take a closer look at a few image hashing techniques and describe them step by step. Work covers such important topics as the properties of the cryptographic and perceptual hash functions, distance/similarity metrics and content processing operations. The paper gives answers why popular cryptographic hash functions do not suit for perceptual image hashing, why other approaches are needed and how they can be implemented.

The thesis consists of four parts, each describing its own objective:

1. Review cryptographic and perceptual hash functions, their properties and give examples of various applications scenarios.
2. Provide different distance/similarity metrics for the evaluation of robustness and the discriminative capabilities of image hashing schemes.
3. Propose the perceptual image hashing framework and describe all required steps to produce the final hash.
4. Describe three basic perceptual image hashing functions, give an overview of some other existing algorithms and propose the practical implementation of the described methods in MATLAB.

The main conclusions are related to the importance of the perceptual image hashing; for example why it is more superior for image comparison instead of traditional cryptographic hash functions. Furthermore, the described perceptual hash functions were compared to verify the robustness and discriminative properties of the each and final results were listed in the overview table. The best of the described algorithms is also proposed in the conclusion, based on the provided overview under different CPO attacks and comparison in case of perceptually different images.

This thesis is written in English and is 70 pages long, including 3 chapters, 32 figures and 7 tables.

Annotatsioon

Piltide pertseptuaalse räsimise rakendamise metoodikaid

Antud töö eesmärgiks on arutada piltide pertseptuaalse räsimise teoreetilisi aluseid, tuues esile piltide räsimise raamistiku ja uurides olemasolevaid piltide pertseptuaalse räsimise funktsioone. Uuring näitab peamisi erinevusi pertseptuaalse ja krüptograafilise räsimise funktsioonide ja rakenduses kasutatavate variantide vahel. Samuti analüüsib autor mõningaid piltide räsimise tehnikaid ja kirjeldab neid sammude kaupa. Töö puudutab olulisi teemasid, nagu krüptograafilise ja pertseptuaalse räsimise funktsioonide omadusi, kauguse/sarnasuse mõõtmeid ja sisu töötlemisoperatsioone. Lõputöö annab vastuseid küsimustele, miks ei sobi populaarsed krüptograafilised räsimisfunktsioonid piltide pertseptuaalse räsimise jaoks ning miks on vaja kasutada teisi lähenemisi, ja kuidas oleks neid võimalik implementeerida.
Lõputöö koosneb neljast osast, kus iga osa kirjeldab oma eesmärki:

1. Ülevaadata krüptograafilise ja pertseptuaalse räsimise funktsioone koos nende omadustega, ja tuua näited, mis demonstreeriksid piltide räsimisskeemasid.
2. Esitada erinevaid kauguse/sarnasuse mõõtmete jõulisuse ja piltide räsimisskeemade diskrimineerivate võimsuste hindamiseks.
3. Esitada piltide pertseptuaalse räsimise raamistiku ja kirjeldada kõiki vajalikke samme, et valmistada lõplikut räsi.
4. Kirjeldada kolm põhilist piltide räsimisfunktsiooni, anda ülevaadet teistest olemasolevatest algoritmidest ja pakkuda MATLAB-i abil kirjeldatud meetodite praktilisi implementatsioone.

Põhilised järeldused on seotud piltide pertseptuaalse räsimise tähtsusega; näiteks, miks see sobib piltide võrdluseks paremini kui traditsioonilised krüptograafilise räsimise funktsioonid. Järgnevalt olid võrreldud kirjeldatud pertseptuaalse räsimise funktsioonid, et võrrelda jõulikuse ja diskrimineerivaid omadusi, ja lõplikud tulemused olid nimetatud ülevaate tabelis. Kõige paremini kirjeldatud algoritm on samuti pakutud järelduses, mis on tehtud esitatud CPO rünnakutest ülevaate ja pertseptuaalselt erinevate piltide võrdluse alusel.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 70 leheküljel, 3 peatükki, 32 joonist, 7 tabelit.

Table of abbreviations and terms

2D	Two Dimensional
3D	Three Dimensional
BER	Bit error rate
CBIR	Content-based image retrieval
CCM	Content changing manipulations
CPM	Content preserving manipulations
CPO	Content processing operations
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
ED	Euclidian Distance
FPR	False Positive Rate
HD	Hamming distance
HVS	Human visual system
IDCT	Inversed Discrete Cosine Transform
MAC	Message authentication codes
NHD	Normalized Hamming distance
PHF	Perceptual Hash Function
TPR	True Positive Rate

List of figures

Figure 1. Hamming distance using MATLAB. 23

Figure 2. Normalized Hamming distance using MATLAB..... 25

Figure 3. Bit error rate using MATLAB. 26

Figure 4. Euclidean distance using MATLAB. 27

Figure 5. Activity diagram of the perceptual image hashing framework. 29

Figure 6. Schema of obtaining the hash from the image. 30

Figure 7. Schema of pre-processing operations in the hash generation. 30

Figure 8. Example of different CPO on the image “Baboon”. 32

Figure 9. Pre-processed version of the image “Baboon”. 34

Figure 10. Pre-processing step using MATLAB before feature extraction. 34

Figure 11. Difference of using Gaussian filter after resizing (left) and before (right). .. 35

Figure 12. Schema of feature extraction in the hash generation. 36

Figure 13. Schema of post-processing operations in the hash generation. 37

Figure 14. Pre-processed image “Baboon” and the average color (126.84). 38

Figure 15. Luminance matrix of “Baboon” before applying Average Hash. 39

Figure 16. Binary matrix of “Baboon” after applying Average Hash. 39

Figure 17. Feature vector after applying Average Hash on image “Baboon”. 40

Figure 18. Image comparison before and after applying Average Hash. 40

Figure 19. Getting binary matrix in MATLAB based on Average Hash..... 40

Figure 20. Pre-processed image “Baboon” before applying Difference Hash. 42

Figure 21. Luminance matrix of “Baboon” before applying Difference Hash. 42

Figure 22. Binary matrix of “Baboon” after applying Difference Hash. 43

Figure 23. Feature vector after applying Difference Hash on image “Baboon”..... 43

Figure 24. Example, of feature vector as image before and after Difference Hashing... 43

Figure 25. Getting binary matrix in MATLAB based on Difference Hash. 43

Figure 26. Discrete Cosine Transform (DCT) in MATLAB. 46

Figure 27. Discrete Cosine Transform of the image “Baboon”. 46

Figure 28. DistanceMetrics class. 53

Figure 29. AverageHash class. 54

Figure 30. DifferenceHash class. 55

Figure 31. DCTBasedHash class. 56

Figure 32. Average Hash behavior in case different images and CPO. 61

Figure 33. Difference Hash behavior in case different images and CPO. 62

Figure 34. DCT Based Hash behavior in case different images and CPO. 63

List of tables

Table 1. Examples of finding the Hamming distance. 23

Table 2. Examples of finding the Normalized Hamming distance. 24

Table 3. Examples of finding the Bit error rate (BER). 25

Table 4. Examples of calculating the Euclidean distance. 26

Table 5. Content-preserving and content-changing manipulations. 31

Table 6. Robustness comparison of Average Hash and Difference Hash. 44

Table 7. Robustness comparison of Difference Hash and DCT Based Hash. 47

Table of contents

Introduction 11

Background and motivation 11

Aim and Objectives 12

Outline 12

1. Cryptographic & Perceptual Hash Functions 13

1.1. Cryptographic Hash Functions 13

1.1.1. Application Scenarios 14

1.1.2. Properties 15

1.2. Perceptual Hash Functions 16

1.2.1. Application Scenarios 16

1.2.2. Properties 17

1.3. Comparison in Image Hashing 20

2. Distance/Similarity Metrics for Perceptual Hashes 22

2.1. Hamming distance 22

2.2. Normalized Hamming distance 24

2.3. Bit error rate 25

2.4. Euclidean distance 26

3. Perceptual Image Hashing 28

3.1. Framework 28

3.1.1. Pre-processing 30

3.1.2. Feature extraction 36

3.1.3. Post-processing 37

3.2. Perceptual Image Hash Functions 37

3.2.1. Average Hash 38

3.2.2. Difference Hash 41

3.2.3. DCT Based Hash 45

3.2.4. Other algorithms 48
3.3. Summary 50
4. Implementation in MATLAB 52
4.1. Structure 52
4.2. Distance/similarity metrics 53
4.3. Perceptual image hash functions 53
4.3.1. Average Hash 54
4.3.2. Difference Hash 55
4.3.3. DCT Based Hash 56
References 59
Appendix 1 – Additional Figures 61
Appendix 2 – Code in MATLAB 64

Introduction

Background and motivation

The fast technological advancement has led to the significant increase in multimedia information, especially in the recent years. The total amount of digital content has grown exponentially, and this number is still growing. We live in the world where almost everything can be digitalized and sent from one part of the world to another in just a matter of seconds. However, due to increasing availability of multimedia data in digital form, the number of tools to manipulate digital multimedia has also significantly increased. The content processing operation (CPO) became trivial, and many users of such tools modify the multimedia content almost each day. This tendency has lead to the need of finding the ways of authentication multimedia content to ensure trustworthiness or find the ways of identification.

For this reason, many different authentication techniques have emerged to verify content integrity and prevent forgery [7]. But, of course, to decide whether of not the object is authentic or not we must take into account the type of that object. For example when authenticating an executable file in most cases it is essential that every single bit matches exactly the original one. For such tasks traditional cryptographic hash functions like MD5 or SHA-1 are very well suited and enable to determine whether or not the compared objects are the same. But those are extremely sensitive, and 1-bit change in the input changes the output dramatically (also known as the “avalanche effect”) [9]. Can we use traditional crypto-hashing to meet the integrity and authentication requirements of digital images if considering an image as a data stream?

There are, however, several reasons that impede the direct use of cryptographic techniques for solving multimedia security problems. Unlike textual data that is transmitted through a lossless medium, multimedia data like images (or audio, video, etc.) may be transmitted and stored using a lossy medium to save bandwidth and storage space [19]. Therefore, using traditional cryptographic hash functions for integrity verification or authentication of multimedia content has a problem that a single bit change in the content will significantly change the hash value. To analyze multimedia content, other hashing algorithms are preferable.

Furthermore, humans can easily distinguish multiple images and say whether or not those are the same. A computer, however, sees everything in a very different perspective, so this easy task for a human is quite complicated for a computer. Multiple images can have different digital representations but from the human perception they will all look the same. That leads us to the problem that multimedia content can be illegally distributed unnoticeably for the search algorithms if considering only the cryptographic hashing techniques. Data such as digital images can undergo various manipulations such as compression and enhancement. It makes impossible for traditional crypto-hashing to spot modified pictures and state that those came from the same source.

Aim and Objectives

The aim of this work is to discuss the practical applications of perceptual image hashing and try to answer why cryptographic hash functions do not suit for image hashing, and different method must be used. The whole research on this topic is divided into four objectives. The first, to compare cryptographic and perceptual hash functions, their properties, and various applications scenarios. The second, to provide different distance/similarity metrics for the evaluation of robustness and the discriminative capabilities of image hashing schemes. The third objective, to propose the perceptual image hashing framework and describe it. And lastly, the fourth objective is to describe three basic perceptual image hashing functions step by step and implement those using MATLAB.

Outline

The thesis consists of four chapters. The first chapter reviews cryptographic and perceptual hash functions, gives examples of various application scenarios and the properties of those hash types. The second chapter provides different distance/similarity metrics that can be used to evaluate the robustness and the discriminative capabilities of image hashing schemes. The third chapter proposes the perceptual image hashing framework and describes all required steps to produce the final hash. Also, it gives an overview of how image pre-processing can be done before the feature extraction. The fourth chapter describes three basic perceptual image hashing functions and also provides an overview of some other existing algorithms. Practical implementation of the described methods in MATLAB is also being proposed and outlined in the fourth chapter.

1. Cryptographic & Perceptual Hash Functions

The term hash functions originate historically from computer science, where it denotes a function that compresses a string of some uncertain size to a string of fixed length [26]. The data to be encoded is often called as the message, and the hash value is sometimes referred as the message digest or simply digest [10]. For this purpose many distinct types of hash functions exist, but in most cases they can be divided into two main groups: *cryptographic* and *perceptual* hash functions. However, it should be noted that the objective of a cryptographic hash function and perceptual ones are not exactly the same. Even though, sometimes the area where they are used can intersect, perceptual hash functions are entirely different concept compared to traditional cryptographic hash functions.

This chapter covers the essential differences between both mentioned hash function (cryptographic and perceptual) types as well as their basic application scenarios. Furthermore, the fundamental properties of each described hash function type will be given with some explanations to summarize the core aspects of both in terms of image hashing.

1.1. Cryptographic Hash Functions

Traditional cryptographic hash functions like MD5 or SHA-1 have only one particular goal: convert the source input (message) into a fixed-length bit string (hash). Very often, they are referred as (digital) fingerprints, checksums, or just hash values even thought all serve a slightly different purpose. There are, however, several reasons that actually impede the direct use of cryptographic techniques for solving multimedia security problems

The cryptographic hash functions are considered to be practically impossible to invert, that is, to recreate the input data from its hash value alone. Of course, such statement about the inversion is not always true. In most cases, however, such cryptographic hash functions that cannot be inverted are often referred as one-way hash functions and are considered to be “the workhorses of modern cryptography” [1]. Such behavior deeply relies on the avalanche effect, where a tiny change in input value modifies the output

value drastically. This results in a completely different hash even if only the one bit of information was changed.

Traditionally, data integrity issues are addressed by cryptographic hashes or message authentication functions. The simplest way to provide data authentication mechanism is to employ directly a traditional cryptographic hash function. A data authentication scheme using a cryptographic approach can be primarily categorized into two types according to the data integrity criteria: hard authentication and soft authentication. A hard authentication approach does not allow any bit changes on the input data, like a standardized cryptographic scheme such as message authentication code (MAC) or digital signature standard (DSS). To do this, a hard authentication scheme generates an authentication code from the input data and delivers it along with the data. During the verification process, the authentication code recalculated from the received data are expected to match exactly with the received authentication code as long as the received data or authentication code are not corrupted or manipulated in transit.

Furthermore, with cryptographic hashes, the hash values are random. The message that is used to generate the hash acts like a random seed so that the same data will cause exactly the same result, but the different message will produce an entirely different hash.

The cryptographic hash functions only tell you two primary things: if the hashes are different, then the data is different or vice-versa. As a result, the message integrity can only be validated when every bit of the message is unchanged [9]. Such behavior is the essential feature of this kind of hashing – ***extreme sensitivity to the input message*** and numerous application scenarios of cryptographic hash functions are based on this behavior.

1.1.1. Application Scenarios

Cryptographic hash functions have many various applications. In most cases, however, they are used in the context of *information security* and in *authentication* where every bit of the compared messages need to be unchanged to verify their integrity. Popular hash functions like MD5 or SHA-1 have many different application scenarios in digital signing or message authentication codes (MACs).

The main application scenario that first comes to the mind when using traditional cryptographic hash functions – ***verifying the integrity of messages***. For this case such

cryptographic hash algorithms like MD5, SHA-1 or SHA-2 are often being used. To determine if there were any changes made in the message (source), two hashes are usually generated: for the original message and the one that needs integrity verification. If the message that is being verified has even at least one single bit of information change, then the resulting hash will be entirely different from the original one. In addition, the same approach is used for ***verifying the integrity of files*** where the content of the file is considered to be as the input message when generating the hash. Moreover, the same concept can go for ***files or data identification***. For example it is widely used in the source management systems like Git or Mercurial for identification of file content, changes, etc. to uniquely identify them.

Another widely used application is the ***password verification***. Instead of storing the original user passwords in the databases only the hashes of those are being stored and later compared with the generated hash from the password user uses during the authentication process.

1.1.2. Properties

Cryptographic hash functions must preserve *three* main properties in order to ensure that they are efficient enough from the perspective of security and can withstand all known types of cryptanalytic attack [8]:

1. Pre-image resistance
2. Second pre-image resistance
3. Strong collision resistance

Pre-image resistance:

Also known as *"one-way"* function and means that for any given hash value h, it should be hard and even impossible to find a message m that would map to this output $h = H(m)$ (where H represents the cryptographic hash function that produces the final hash). This means that it should be easy to take a message m and generate the hash h value from it, but it is impossible to take a hash h value to re-create the original message m.

Second pre-image resistance:

Second pre-image resistance is also very often referred as *"weak collision resistance"*. Given a hash value h and a corresponding message m_1, it should be impossible to find another message m_2 (where $m_1 \neq m_2$) with the same hash value $H(m_1) = H(m_2)$ (where H represents the cryptographic hash function that produces the final hash).

Strong collision resistance:

It should be infeasible to find two distinct messages $m_1 \neq m_2$ that will produce the same resulting hash values $H(m_1) = H(m_2)$.

All three properties imply that a malicious adversary cannot replace or modify the input data without changing its digest. The level of security, however, of each property relies on various ways of how those are achieved. The

1.2. Perceptual Hash Functions

Perceptual hash functions (PHF) are an entirely different concept compared to the traditional cryptographic ones. However, like a cryptographic hash function, a perceptual hash function is designed to take some message as an input and produce a fixed-length output, which is also called as a hash value or message digest. But in this case, the main idea is not to be very sensitive to the individual bits in the message but instead ***be sensitive to the differences in the "perceptual features"*** of the message. Unlike cryptographic hashing functions, the perceptual ones give you a sense of similarity between the two data sets. Unlike, the traditional cryptographic hash functions which produce the hash based on the content of the message, the perceptual hash functions generate the hash based on the "feature vector" extracted during the "feature extraction" process. In other words, perceptual hash functions generate the hash based on the distinctive features of the multimedia object instead of the data stream by itself. The message, however, can be any multimedia object, but for feature extraction it will require entirely different algorithms.

Because the aim of this work is the perceptual image hashing let's refer to the multimedia object as image. The various application scenarios or main properties of perceptual hash functions will not change even if other multimedia objects are used. The only differences will be in the feature extraction methods used to produce the feature vector.

1.2.1. Application Scenarios

There are many practical ways of using PHFs. An immediately obvious one is ***identification/search of multimedia object in large databases***. It can be used search engines to find copyright violations or to maintain a database of illegal content like child pornography. At the moment of writing this work, there were many existing solutions that can identify illegal content by having a large database of hashes:

- *EnCase Forensic* from Guidance Software
- *FTK* (Forensic Toolkit) from Access Data
- *X-ways* from Software Technology
- *DFF* (Digital Forensics Framework)

But if speaking about the images, those tools can be easily fooled just by slightly modifying the original images in order to change their cryptographic hashes. The perceptual hashing will be an excellent addition to that kind of software and furthermore will eliminate the need of having multiple hashes of the same images that are identical from the human perspective.

Another practical use of image hashing is the ***content-based image retrieval*** (CBIR). It leads to very many opportunities for improving image search algorithms in the Internet, automate/simplify the process of making photograph archives or retail catalogs, integrate nudity detection filters etc.

In addition to what was already mentioned perceptual image hashing can improve the ***image watermarking*** techniques that can be used for image authentication. The fundamental idea of watermarking is to embed invisible secondary data (watermarks) depending on the application, onto the digital images. As a result, all copies of the images containing the watermark can be used to tell whether the image is authentic or not and prove the ownership. The "perceptual features" of the image can be included in the watermarks that will not only increase the security of it but also can make much easier validity verification.

1.2.2. Properties

Perceptual hash functions must preserve *four* main properties in order to ensure that it is efficient and secure at the same time:

1. Robustness

2. Discriminability
3. Unpredictability
4. Compactness

In order to easier get the idea let's assume that O is the original image and $\hat{O}$ is a modified version of it but preserves the perceptual properties of the original one (O). In other words, the O and $\hat{O}$ are the same images from the human perspective but different in cryptographic way. The M stands for a completely different image from the original one (O). Let θ_1, θ_2 two positive values that satisfy $0 < \theta_1, \theta_2 < 1$. The perceptual hash function H produces a hash fingerprint with the length depending on a secret key K [6].

Robustness:

Robustness means that if the same key is used, *perceptually similar images generate a similar hashes* [12]. This property can be represented as the equation:

$$P(H(O,K) = H(\hat{O},K)) \approx 1 \quad (1.1)$$

In Equation 1.1, P denotes probability. H is the perceptual hash function that produces the final hash based on the input image O and the secret key K. The $\hat{O}$ is the modified version of the image with preserved perceptual properties of the original one (O). The hash values of perceptual similar images should be the same or with very small similarity distances, even if the modified image $(\hat{O})$ was changed. In this case, the probability value should be equal or very close to 1.

The robustness ensures that two perceptually identical images should have similar hashes. The modified version of the image can have compression artifacts, noise, and some other distortions. The main purpose of perceptually similar images is to be robust enough to the different CPOs like JPEG lossy transformation, rotation, noise, blurring etc. Perceptually, these images in human visual system (HVS) are identical, even though some of the bits were changed. The perceptual robustness of image hashing guarantees that these images have very similar hashes.

Discriminability:

Discriminability means that if the same key is used, *perceptually different images generate the different hashes*. This property can be represented as the equation:

$$P(H(O,K) = H(M,K)) \approx 0 \quad (1.2)$$

In Equation 1.2, P denotes probability. H is the perceptual hash function that produces the final hash based on the input image O and the secret key K. The M is the completely different image. The hash values of different images should not be the same or with high similarity distance. In this case, the probability value should be equal or very close to 0.

The discriminability ensures that two perceptually distinct images should have different hashes. In other words, the hashes of two completely different images should not be equal. There should be a very low probability close to 0 that two distinct images will produce the same hashes.

Equal distribution of hash values (unpredictability):

Unpredictability ensures that the attacker will not get the *same hash for the original multimedia object by manipulating some of the modified multimedia object data bits*.

$$H(O,K); f_h(1) \approx f_h(0) \approx 0.5 \quad (1.3)$$

In Equation 1.3, H is the perceptual hash function that produces the final hash based on the input image O and the secret key K. Where f_h is the probability mass function for hash h. With this property, the hash values should be equally distributed.

Security is an important concern for hashing. This will make the procedure of hash generation secure enough to decrease the probability for the attacker to guess the secret key and estimate the correct hash value. For perceptual image hashing traditional cryptographic hash functions can be used or other different pseudo-randomization techniques can be incorporated into the image hash generation process by using secret keys [6].

Compactness:

Compactness is another important property of almost any hashing scheme. This property can be represented as the equation:

$$Size(H(O,K)) \ll Size(O) \quad (1.4)$$

In Equation 1.4, $Size$ represents the amount of total bits. H is the perceptual hash function that produces the final hash based on the input image O and the secret key K.

The size of the hash if comparing to the original image should be significantly smaller. Compactness property solves the problem of having large databases and simplifies the process of searching. Furthermore, there is no need to restore the original image from the hash, but only take into account the "perceptual features" of it.

1.3. Comparison in Image Hashing

A hash provides a compact representation of any data. Different kinds of hash functions have their characteristics and purposes. Even though, sometimes the area where they are used can be the same, each hash function type has its unique purposes. The traditional cryptographic hash functions are very sensitive to the input data where 1-bit change can cause an avalanche effect resulting in an entirely different hash. On the other hand, the perceptual hash functions are not very sensitive to small changes in the input data but are susceptible to the differences in the "perceptual features" of the compared object.

Because of the extreme sensitivity to the input message, two cryptographic hashes can only be compared to determine whether or not they came from the same source. We cannot capture and measure the similarity them to ascertain if the sources are distinct or not. Traditional cryptographic hash functions only tell us one basic thing: if the input data is different or not. Such behavior results in various application scenarios in the context of information security and authentication, where even single bit change matters. But we can not use traditional crypto-hashing and digital signatures to meet the integrity and authentication requirements of digital images. Even though, it is still the simplest way to authenticate a digital image is by using the traditional cryptographic hash functions like MD5 or SHA-1 along with public key encryption algorithms such as the RSA [9].

As was already discussed above, the problem of using cryptographic hash functions for image hashing is the avalanche effect. For images, however, it is very common to undergo different CPO, lossy medium, compression, channel noise, etc. that will not change the image for HVS and will result in entirely different hash. The perceptual hash functions serve this goal much better. For this purpose, they are resilient to content preserving manipulations like compression, channel noise, etc., and at the same time are fragile enough to detect malicious manipulations [19]. Unlike cryptographic hash functions, they allow us to make comparisons between hashes to measure the similarities between the

sources and give us the sense of how “distant” they are. In the Chapter 2, we will look more closely on some of the similarity metrics for measuring the distance.

However, due to the fact that the cryptographic hash functions are matured and well studied for more than a decade, it is a real challenge to design perceptual image hash functions that besides meeting the requirements of multimedia applications follow the security features of the first one [21]. To solve this problem, many studies were made. The one of the easiest ways of getting the secured version of the image hash is to use the one of the traditional cryptographic hash function alongside with the perceptual one in order to encrypt it as the final measure. But if talking solely about the image comparison task, then the cryptographic hash functions are not suited, and the use of only perceptual hash functions is encouraged.

2. Distance/Similarity Metrics for Perceptual Hashes

To evaluate the *robustness* and *discriminative* properties of the image hashing different distance/similarity metrics are required that will show the differences between two similar media objects. It is crucial in the image (or mostly for any other multimedia object) hashing because based on that we can claim if the images are the same or entirely different. When the hash is being generated from the image that needs to be verified whether or not it is in the database, the distance between each image must be calculated and compared. In other words, two hashes must show the distance of how "perceptually different" those images are. For this purpose, *three* most popular metrics is proposed that can be used for this task and are being described in this chapter.

For better understanding the logic of each distance calculation technique, two hashes can be represented as Equations 2.1 and 2.2:

$$H_1 = h_1(1), h_1(2), \dots, h_1(L) \tag{2.1}$$

$$H_2 = h_2(1), h_2(2), \dots, h_2(L) \tag{2.2}$$

where H_1 and H_2 are the hashes of two different images with particular hash length L. The hashes represent an array of bits h. Depending on whether or not we are dealing with binary or decimal hashes, we can use different approaches of distance calculation.

2.1. Hamming distance

The Hamming distance (HD) measures two binary hash vectors similarities by comparing those bit-by-bit and producing the number of bits which differ. It measures the minimum number of substitutions required to change one string into the other [4]. For example let's assume that we have two binary strings (A and B), then the Hamming distance will be equal to Equations 2.3:

$$HD = \sum |A_i - B_i| \tag{2.3}$$

where HD represents the Hamming distance, A is the first binary string and B is the second. The higher the result of Hamming distance, the more differences the compared strings have. In our case the string are hashes. If the Hamming distance of two hashes

equals 0, then the hashes are the same. The higher the result, the more differences two hashes have and they are more "distant" from each other. However, the Hamming distance cannot be more then the length of the string. Sometimes the number of characters is used instead of the number of bits. This means that it can consist of elements from alphabets or other number systems (Table 1).

Table 1. Examples of finding the Hamming distance.

A and B are the strings. Examples are given for binary system, Latin alphabet and decade system.

A	B	Hamming distance
0100100	0100100	0
0100**1**0**0**	**1**100**1**1**0**	2
k**a**r**ol**in	k**e**r**st**in	3
43**9**125**6**	**5**3**4**125**5**	3

Even though, as was shown in Table 1, even decade system or alphabet can be used, for our purpose only binary representation of compared data is needed when calculating the Hamming distance. Taking this into account, we can make an equation using XOR operator ($\oplus$). In other words, we can denote a and b as two binary coded numbers of equal length. Then the Hamming distance will be equal to the number of ones in a $\oplus$ b [13]. By expanding and slightly adapting the basic Equation 2.3 we can get the full equation of Hamming distance for two binary hashes (Equations 2.4):

$$HD(H_1, H_2) = \sum_{i=1}^{L} |h_1(i) \oplus h_2(i)| \tag{2.4}$$

where HD represents the Hamming distance. H_1 and H_2 are the binary hashes of two different images with the same length L. h_1 is the first binary hash bit, h_2 is the second. For Hamming distance, the length of the both examined strings must be equal.

In MATLAB the Hamming distance can be calculated using pairwise distance between two sets of observations (binary vectors with the same length) known as "pdist2" function and multiplied by the length of the first strings (Figure 1):

```
HD = pdist2(A, B, 'hamming') * length(A);
```

Figure 1. Hamming distance using MATLAB.

where HD is the resulting variable of obtaining the Hamming distance. A and B are the two binary vectors equal in length. "pdist2" is the pairwise distance function between two sets of observations with metric specified as "hamming". The result of "pdist2" must be multiplied by the length of one of the strings.

2.2. Normalized Hamming distance

The Hamming distance can be normalized with respect to the length L of the strings (Equation 2.5). Normalized Hamming distance is especially useful when we want to get the ratio of the elements that differ. In Table 2 are given the examples of getting the Normalized Hamming distance for binary system. By adapting the Equation 2.4 for Hamming distance, we can get the normalized version of it (Equations 2.5):

$$NHD(H_1, H_2) = \frac{1}{L}\sum_{i=1}^{L} |h_1(i) \oplus h_2(i)| \tag{2.5}$$

where HND represents the normalized Hamming distance. H_1 and H_2 are the binary hashes of two different images with the same length L. h_1 is the first binary hash bit, h_2 is the second. For Hamming distance, the length of the both examined hashes must be equal.

For Normalized Hamming distance, the result is expected to be close to 0 for similar images and close to 0.5 for dissimilar ones. As more parts of a picture are changed, the manipulated image and the original image become more divergent. For an ideal hashing scheme, the normalized Hamming distance between the corresponding hashes should increase accordingly [6]. The maximum value of the normalized Hamming distance will be 1, but it is very unlikely that it will ever reach it in case of image hashing. That is the cause of why the value close to 0.5 is more truthful for dissimilar images.

Table 2. Examples of finding the Normalized Hamming distance.

A and B are the binary strings. Examples are given for binary system.

A	B	Normalized Hamming distance
0100100	0100100	0
0100**1**0	**1**100**1**0	0.286
01**00**1**00**	**1**1**11**1**11**	0.714

0100100	**1011011**	1

In MATLAB the Normalized Hamming distance can be calculated using pairwise distance between two sets of observations (binary vectors with the same length) known as “pdist2” function (Figure 2):

```
NHD = pdist2(A, B, 'hamming');
```

Figure 2. Normalized Hamming distance using MATLAB.

where NHD is the resulting variable of obtaining the Normalized Hamming distance and A and B are the two binary vectors equal in length. “pdist2” is the pairwise distance function between two sets of observations with metric specified as “hamming”. The final result will be the Normalized Hamming distance.

2.3. Bit error rate

Another very widely used distance metric is the Bit error rate also known as BER. In Table 3 are given the examples of getting it for binary system. BER can be represented as the following equation (Equations 2.6):

$$\rho := \frac{i}{L} \tag{2.6}$$

where ρ represents Bit error rate. i is the number of bit errors of the hash normalized by its length L. Furthermore, Bit error rate must be in the following range $0 \leq \rho \leq 1$ and $i \in \{0, 1, 2, \dots, L\}$. The result of the number of bit errors i is the same as the Hamming distance between two different hashes and the final result equals to the Normalized Hamming distance.

Table 3. Examples of finding the Bit error rate (BER).

A and B are the binary strings. Examples are given for binary system.

A	B	BER
0100100	0100100	0
0100**1**0**0**	**1**100**11**0	0.286
0100100	**1111111**	0.714
0100100	**1011011**	1

Perceptually two identical images should have a value of BER close to 0, otherwise it should be approximately 0.5 [13]. Like with the Normalized Hamming distance, it is very unlikely that the Bit error rate will ever reach the maximum value of 1 for image hashing task, so the value 0.5 is more accurate for the dissimilar images.

In MATLAB the Bit error rate can be calculated using "biterr" function (Figure 3):

```
[num_bit, ratio_bit] = biterr(A, B);
```

Figure 3. Bit error rate using MATLAB.

where "num_bit" is the total number of bit errors and "ratio_bit" is the Bit error rate. A and B are the two binary vectors equal in length.

2.4. Euclidean distance

The common Euclidean distance (square root of the sums of the squares of the differences between the coordinates of the points in each dimension) serves for all Euclidean spaces [17]. It is suitable for non-binary vectors like integers. It is perfect for measuring similarity and discriminating capability between two hashes when it is not represented in the binary form. In Table 4 are given the examples of getting it for binary system. The Euclidean distance can be represented as the following equation (Equations 2.7):

$$ED(H_1, H_2) = \sqrt{\sum_{i=1}^{L} \big(h_1(i) - h_2(i)\big)^2} \tag{2.7}$$

where ED represents the Euclidean distance. H_1 and H_2 are the decimal hashes with the same length L. h_1 is the first decimal hash bit, h_2 is the second. For Euclidean distance, the length of the both examined hashes must be equal. The lower the Euclidean distance is, the closer the two hash values are.

Table 4. Examples of calculating the Euclidean distance.

A and B are the strings. Examples are given for binary system and decade system.

A	B	Euclidean distance
0100**1**0**0**	**1**100**1**1**0**	1.414
43**9**125**6**	**5**3**4**125**5**	5.196

In MATLAB the Euclidean distance can be calculated using pairwise distance between two sets of observations (decimal vectors with the same length) known as “pdist2” function (Figure 4):

```
ED = pdist2(A, B, ' euclidean');
```

Figure 4. Euclidean distance using MATLAB.

where ED is the resulting variable of obtaining the Euclidean distance and *A* and *B* are the two decimal vectors equal in length. “pdist2” is the pairwise distance function between two sets of observations with metric specified as “hamming”. The final result will be the Normalized Hamming distance.

3. Perceptual Image Hashing

As was already stated during the comparison of traditional cryptographic and perceptual hash functions in Chapter 1.3, a simple way to authenticate a digital image is to use a cryptographic hash function like the MD5. However, the ways of how two compared hash function types behave are entirely different. While cryptographic hash functions are extremely sensitive to each bit, perceptual hash functions are sensitive only to perceptual features. Such behavior of perceptual hash functions makes them a perfect choice for image hashing and comparison based on the distinctive features of analyzed images.

This chapter proposes the perceptual image hashing framework and describes the main three steps of a hash generation scheme. Furthermore, most widely used perceptual image hash functions are being defined starting from the most fundamental approaches and ending with more advanced one.

3.1. Framework

After reviewing different digital image hashing algorithm designs, we can state that the design framework of digital image hashing depends on various application scenarios and the ways of how secure the hash must be. For example, if the digital image hashing algorithm is designed for content identification, it mainly should concern only the robustness property against different content-preserving manipulation (CPMs) that do not destroy the perceptual quality of the image. On the other hand, image authentication strongly relies on content-changing manipulations (CCMs) in the image content such as removal and object insertion, image hashes should be sensitive to these perceptually significant attacks. Depending on the threshold of the similarity we can variate the accuracy of the algorithm and decide whether we are dealing with the same image or with perceptually similar.

Furthermore, the image hashing framework must consider how secure the whole process of obtaining the final hash can be. It is important because the perceptual image hash function are known and available to public it may result in using feature extraction alone susceptible to forgery attacks, even when the final hash is obtained by encrypting these features. The reason behind it is to decrease the chances for the attacker to create a new image with different visual content while still preserving the perceptual feature values

[6]. As was already mentioned in comparison between cryptographic and perceptual image hashing, for this purpose traditional cryptographic methods like MD5 still can be used to produce the final image hash. However, in most cases the use of cryptographic hashes is not justified because we will not be able to use the similarity metrics in comparison if the hash is irreversible.

By taking into account the features described above and extending the basic idea of hash generation, the whole process of the perceptual image hashing can be generalized as activity diagram in Figure 5.

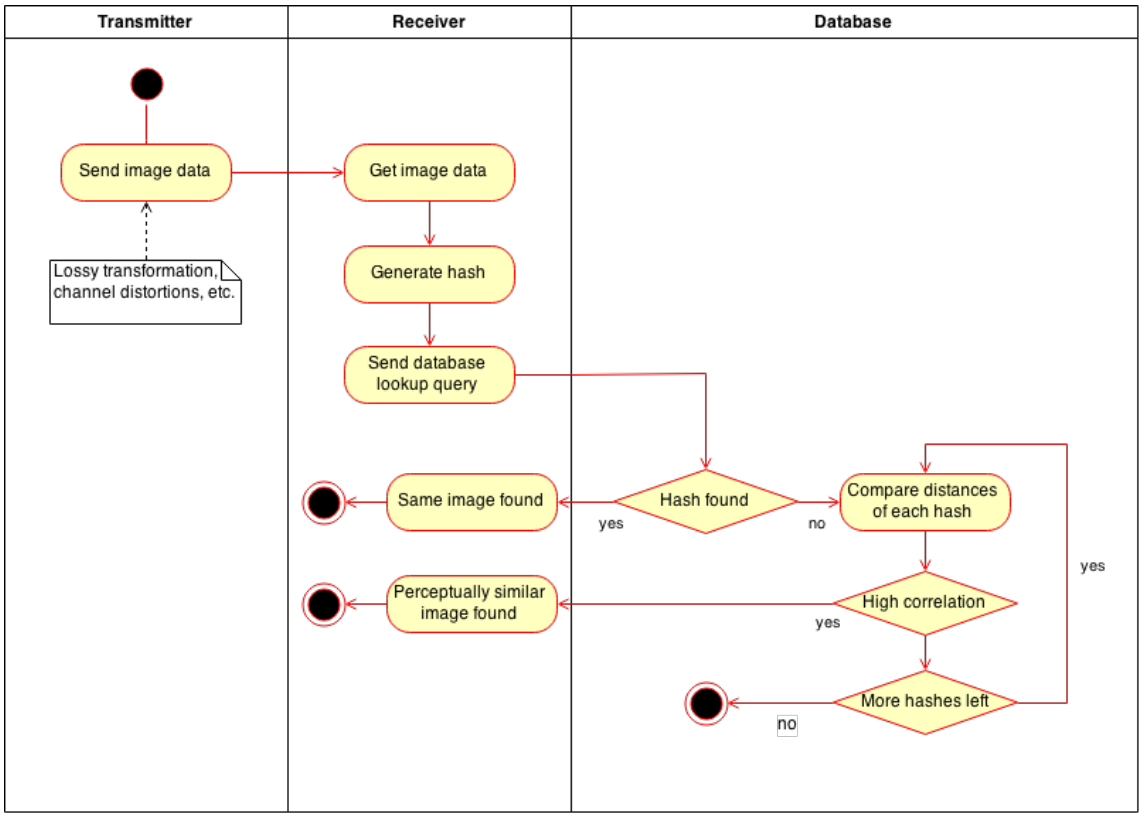


Figure 5. Activity diagram of the perceptual image hashing framework.

As depicted in Figure 5, hash is generated from the image data transmitted to the receiver. The image may be transmitted and stored using a lossy medium to save bandwidth and storage space [19]. Even in this case we will end up having an image that from the traditional cryptographic perspective in most cases will be different because some of the bits will differ. At the receiver's end, the image hash is then generated from the received image and is the database lookup query is sent. If the same hash was found in the database, then we can state that the image was authenticated. On the other hand, if no exact match was found then the distance metrics like Hamming distance can be used in order to find

the hash we the smallest distance or the first occurrence depending on the specified threshold. If the "similarity score" is high (low distance between hashes), then we can state that the image is identified and we are dealing with perceptually similar one. Even though, the exact algorithm of how the system will behave depends strongly on the application scenarios, in most cases the activity diagram of perceptual image hashing will be the same.

Of course, in the provided activity diagram the most important part is the hash generation. The whole process of obtaining the final image hash can be generalized into one simple scheme that consists of three steps: pre-processing, feature extraction and post-processing (Figure 6).

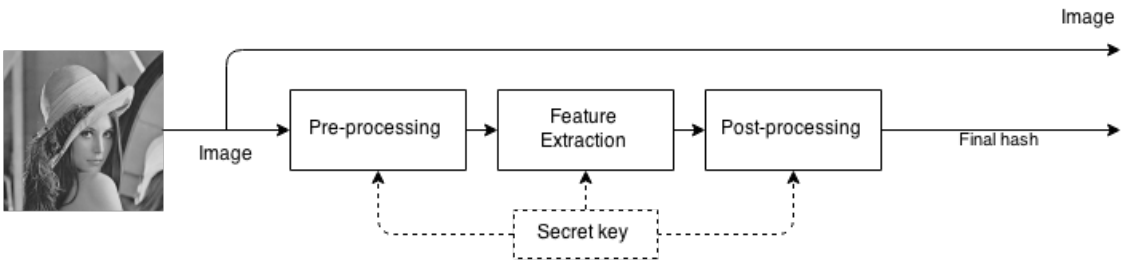


Figure 6. Schema of obtaining the hash from the image.

The robustness property of image hashing comes mainly from the 1st and 2nd steps. It depends on how accurate was the feature extraction process and what additional security measures against different CPO attacks were considered in the pre-processing step. Even though, the security is not the primary goal of this work; the perceptual image hashing framework must consider it and implement at least some basic security features. For this reason, the use of a secret key is proposed in each of the steps as illustrated in Figure 6. The idea is to make each step key-dependent resulting in completely random final hash that still can be used to get the distance metric between two different hashes produced by the same algorithm and the same keys.

3.1.1. Pre-processing

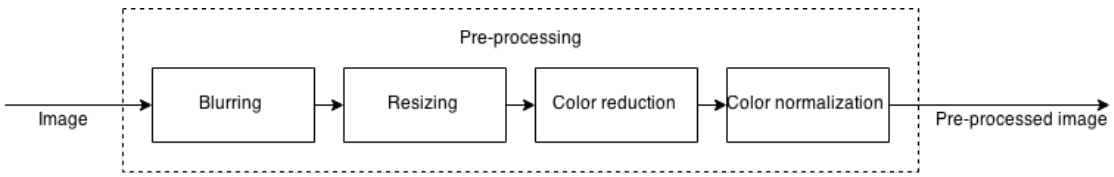


Figure 7. Schema of pre-processing operations in the hash generation.

The main purpose of the pre-processing step (Figure 7) is to prepare the image for the future extraction. The idea is to minimize the amount of total data needed to be processed. In addition to that also increase the robustness of the future extraction to the noise and some other CPO like compression artifacts, etc. [6]. The same thing goes for the image color. The colored version should be transformed to grayscale in order to reduce the need of extra calculations required. Use of the additional color channels doesn't affect accuracy very much in feature extraction but only make the total algorithm heavier.

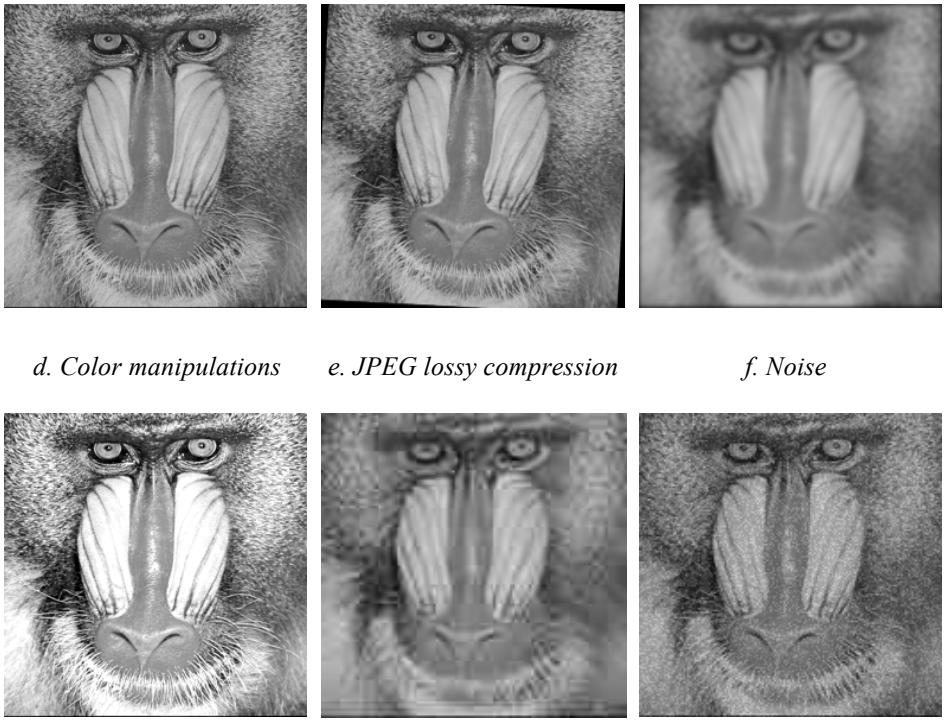
Table 5. Content-preserving and content-changing manipulations.

Content-preserving manipulations	**Content-changing manipulations**
Noise addition	Removing image objects
Resolution reduction	Adding new objects
Compression and quantization	Moving of image elements
Scaling	Changes of image characteristics: color, textures, structure, etc.
Rotation	Changes of the image background: day time or location
Cropping	Changes of light conditions: shadow manipulations etc.
Changes of brightness hue and saturation	
Contrast adjustment	
Transmission errors	

In other words, the frequencies of the processed image must be reduced to the amount when they still preserve robustness property, but make the data required to analyze in the feature extraction step less. In addition to decreasing "calculation cost", the pre-processing step is also needed to standardize the whole process.

In Table 5 are shown all CPO including content-preserving manipulation (CPM) and content-changing manipulation (CCM) that can be applied to the image [19].

a. Original *b. Rotated* *c. Blurred*



d. Color manipulations *e. JPEG lossy compression* *f. Noise*

Figure 8. Example of different CPO on the image "Baboon".

In Figure 8 are given an example of different CPO that that image hashing should be robust enough. The aim of the pre-processing step is to eliminate the differences as much as possible while preserving the properties needed for the image identification. For this action, the most common pre-processing operations that lower the frequencies while increasing the robustness property of an image are:

1. ***Blurring:*** a good way to ensure that the image will be robust enough to blurring distortions and especially strong against noises. For this purpose a traditional *Gaussian filter* can be used, which results in a blurry image and eliminates some of the image contents. It is an efficient way to lower the FPR. The blurring operation is a good way to reduce high-frequency components and alleviate influences of minor image modifications, e.g., noise contamination and filtering, on the hash values.
2. ***Resizing:*** a very common way of reducing the amount of data needed to be processed and at the same time also improve the *robustness* property because it is much easier to extract features from the image with standardized. It also a good and the fastest way to eliminate high frequencies and detail of the image. However, depending on the resulting size the image can have a higher FPR if reduced in the size too much. The most important is to use the bi-cubic

interpolation for image resizing because it the most suitable for the image hashing because it doesn't "throw out" the necessary data that can result in a higher FPR. Furthermore, the image resizing is also a way to ensure that those images with different resolutions have the same or very similar hashes. The resizing also should be $m \times m$ dimensions, to ensure that the resulting hash is going to have a fixed size. In this work let's use the size of 32x32 as the default one.

3. ***Colors reduction:*** the idea is also to lessen the total amount of data needed to be processed that is not required for the task of perceptual image hashing. The color reduction is a very standard for image hashing that provides a good way to increase the performance of the whole process. Change a 3D vector to 2D by removing the red, green and blue channels from the image and converting it to grayscale. In other words, we conduct color space conversion from *RGB* color space to *YCbCr* color space and take the luminance component for image representation [6]. For example, the conversion can be done by the following equation:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 48.512 & 119.317 & 18.25 \\ -37.797 & -51.543 & 124 \\ 124 & -53.124 & 7.543 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \quad (3.1)$$

   where R, G and B are the red, green and blue components of a pixel, and Y, C_b and C_r are its luminance, blue-difference chroma and red-difference chroma, respectively [6].

4. ***Color (Illumination) normalizations:*** this is the way of improving robustness to the brightness and changes in the gamma. For this purpose, a traditional *Histogram equalization* method can be used.

Taking into account all the pre-processing operations that were described above, at the end of the pre-processing step we will end up having a small, grayscale and blurred version of the image with normalized color. All the highest frequencies were removed. If taking and example of the images from Figure 8 we will end up looking with something like this:

Figure 9. Pre-processed version of the image "Baboon".

Size 8x8. On the left is the original one, on the right is the magnified version of it.

The figure above demonstrates the pre-processed version of the original image "Baboon". The size of it is 8x8 pixels. That produces in total the value of 64 bits with a luminance component Y of YCrCb as the value of each bit. The luminance component contains the main structural and geometric information of the image. The example in Figure 9 is the most common way of pre-processing the image before feature extraction, however, it can vary depending on the perceptual hashing function. For example the size can be different or additional filtering method can be applied.

```
% Resizing
img = imresize(img, [8 NaN], 'bicubic');
% Blurring
img = fspecial('gaussian', [3 3], 2);
img = imfilter(img, filter, 'same');
% Color reduction
img = rgb2gray(img);
% Color normalizations
img = histeq(img);
```

Figure 10. Pre-processing step using MATLAB before feature extraction.

In Figure 10 is given the MATLAB example of the pre-processing step based on Figure 7. The most important feature that the pre-processing step must preserve is that if the image were suffering from some small CPO attacks (Figure 8), the image would be normalized. The different CPO will produce the image that will most likely have a very similar values in each bit. At least the distance of each bit from the same perceptually identical images in most cases will not be very high, and the values will not differ very much. Especially that concerns when the one of the compared images had noise, blurring or JPEG lossy artifacts. Even the small rotation of the image will not change the values very much because the bicubic resizing will soften up the small differences.

The order of each operation should not change because it can result in higher FPR. In many cases however the changing order will not affect the final result if the order is

preserved. But it is highly recommended to use it as is, and further research is required to consider which of the cases will have less FPR.

However, the applying of Gaussian filter (blurring) should be considered to be done before the resizing step because it will produce the unwanted "dark border". It can result in higher FPR if the main differences of the images are located near the border. In addition to that, it will also result in some detail loss. Furthermore, the use of the blurring before resizing will also make the image more robust to significant variations of *sigma* value in the Gaussian filter.

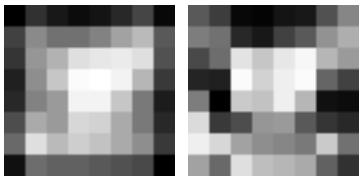


Figure 11. Difference of using Gaussian filter after resizing (left) and before (right).

In the Figure 11 is given an example of the pre-processed and magnified version of the image "Baboon" that was used before as an example in Figure 8. The original size of the image 8x8 pixels. On the left the Gaussian filter with the same properties is applied after resizing, which results in loss of some details and "dark border" appearance. On the right side is the same image but the Gaussian filter is applied before the resizing. This results in the much crisper image and the pixel colors are more distinct from nearby neighbors. Of course, the properties of the Gaussian filter can be adjusted to make an image on the left look more like the one on the right. But, in this case, a problem can appear that the Gaussian filter applied is not noticeably reducing the noise.

But in most cases, the order of whole operations will not dramatically decrease the accuracy because the important part is still will be going in the feature extraction. Depending on different application scenarios and the perceptual image hash functions used the size and the blurring can be adapted to get different accuracy and the amount of data to be processed in the feature extraction step. For example for *Average Hash* this size is considered to be the most optimal [15], but for *DCT* the size of 32x32 is more recommended [14]. The advantage of applying dimensionality reduction techniques are mainly for robustness against noise addition, blurring and compression attacks [6].

3.1.2. Feature extraction

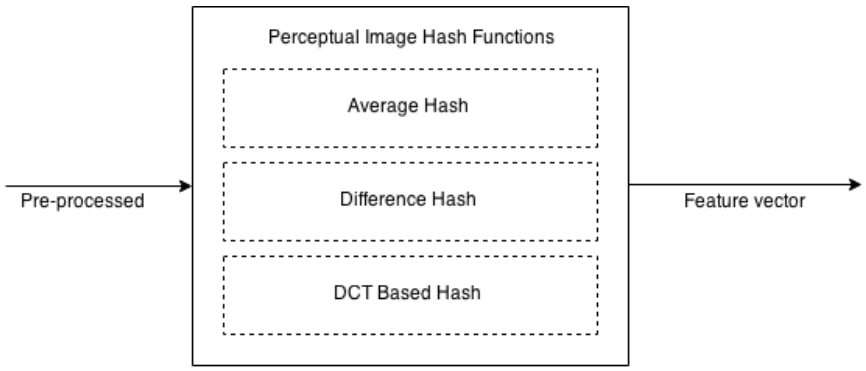


Figure 12. Schema of feature extraction in the hash generation.

The feature extraction step (Figure 12) is the most important. Based on the result it produces we can state if the whole perceptual image hashing algorithm is working correctly or not. This is the primary step required for content identification, and the extracted feature vector must be perceptually similar enough to find the same images and distinctive at the same time to notice the differences. In other words, the data produced in this step for two similar images must have a very little distance based on the proposed metrics in Chapter 2 while keeping the *robustness* and *discriminability* properties.

The feature extraction can be done in many different ways, and there were many studies in this field to find a perfect perfect one. However this is most likely not possible because depending on the different application scenarios where it is used some of the algorithms will not be robust enough for some of the tasks but provide very good "performance score". Or it could be an opposite situation, where the results of feature extractions are good but the speed of calculation is not enough for the use in production.

The feature extraction for the perceptual image hashing is very often named as perceptual image hash functions because those are the core of the whole process. The same naming convention is used in this work, and all the studied perceptual image hash functions are described in Chapter 3.2. Furthermore, in the same Chapter 3.2 pre-processing and post-processing operations are also included, because depending on the perceptual image function used they can slightly differ from each other.

3.1.3. Post-processing

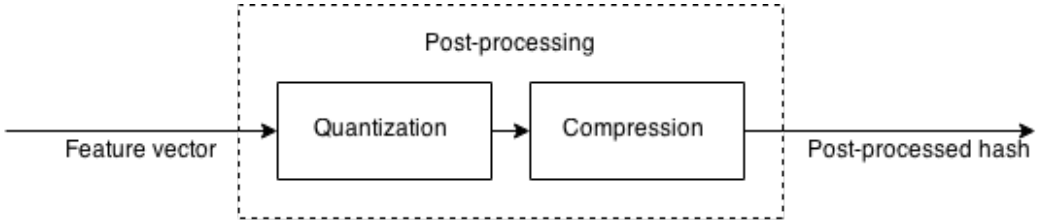


Figure 13. Schema of post-processing operations in the hash generation.

The step of post-processing should not result in any changes in the final hash from the point of view of interfering with the similarity properties of the analyzed image. The aim of this step is only to improve *unpredictability* and *compactness* properties of the perceptual hash function.

Like in the pre-processing chapter, the most post-processing operations are also summarized and the most common are:

- ***Quantization:*** is required to make the resulting hash smaller. The Quantization stage in a perceptual image hashing system is very important to enhance robustness properties and increase randomness to minimize collision probabilities in a perceptual image hashing system [19]. The different popular approaches include interval quantization, binary quantization using ordinal measures or threshold for image hashing generation.
- ***Compression:*** the final step of a perceptual image hashing system, the binary intermediate perceptual hash string is compressed and encrypted into a short perceptual hash of fixed size. This stage can be ensured by cryptographic hash functions i.e. SHA series which generate the final hash of fixed size [20].

3.2. Perceptual Image Hash Functions

The feature extraction step strongly depends on the algorithms which can be used. For image hashing they are often called as perceptual image hash functions. The basic algorithms like Average and Difference Hashes, contain the main idea of how perceptual hashing works. While being fast, they are not very accurate as the DCT Based Hash, even though they work and show quite good results. Depending on different applications scenarios, the methods can vary depending on what is more required: the speed or accuracy.

The Average and Difference Hashes show an excellent performance results ([15] and [16]). However, they will mostly be good for finding similar images with and without watermarks or the thumbnail versions of them. The DCT Based hash is considered to be more complicated than the previously mentioned.

3.2.1. Average Hash

This chapter covers the theoretical discussion on the Average Hash technique which is considered to be the most basic one. However, while being the simplest it still produces quite good results in terms of image comparison.

With pictures, high frequencies give you detail while low frequencies show you structure. A large, detailed picture has lots of high frequencies, unlike the small ones [15]. For this purpose, our proposed hashing framework has the pre-processing step introduced in Chapter 3.1, and it has one single goal: eliminate high frequencies. They ways of how those are removed are the most traditional ones for images: *blurring*, *resizing*, *colors reduction* and *normalization*. For Average Hash the pre-processing operations can be the same as described earlier in Chapter 3.1.1, including all the described operations.

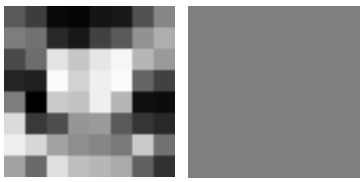


Figure 14. Pre-processed image "Baboon" and the average color (126.84).

As a result, we have ended up with the image that is 8x8 dimensions in size. That size is considered to be the most optimal for the task of image hashing [15], which results in the matrix of 64 bits required to be processed.

As you might already guessed, the primary goal of the Average Hash is to find the average color of all those values by calculating the mean of the matrix. Each value in the matrix is just a luminance component Y of YCrCb, since it contains the main structural and geometric information of the image (Figure 17).

89	61	8	4	20	28	81	138
121	117	40	28	61	85	146	178
73	109	227	194	231	247	186	154
36	28	251	210	239	251	101	69
130	0	202	194	239	178	16	12
219	57	73	150	158	89	53	40
239	215	162	138	138	121	202	113
162	101	223	190	178	170	97	49

Figure 15. Luminance matrix of "Baboon" before applying Average Hash.

By calculating the mean value of all luminance values presented in Figure 15, we can get the average color of the processed image. In this case, the average color will be approximately 127 (126.84). Now, we need to compare each element of the matrix to the average color we got before. As a result of this operation, we should get a new binary matrix with 64 elements where 1 and 0 represent the *true* and *false* values respectively. The same matrix can be represented as an image that is shown in Figure 16, where the 1 is white, and 0 is black.

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	0	0
1	0	1	1	1	1	0	0
1	0	0	1	1	0	0	0
1	1	1	1	1	0	1	0
1	0	1	1	1	1	0	0

Figure 16. Binary matrix of "Baboon" after applying Average Hash.

As you might have already noticed in Figure 16, the result binary matrix has the shape of the original preprocessed image. You can clearly see the shape of the baboon. Now by getting the vector from the matrix starting from the top left and going to the bottom right, we should a 64-bit long hash. The resulting hash can be later compared with other hashes

of perceptually similar images to retrieve the "similarity score" based on the distance metrics like the *Normalized Hamming distance* or *BER*.

```
0000000100000011001111110011110000111100100110001111101010111100
```

Figure 17. Feature vector after applying Average Hash on image "Baboon".

For Normalized Hamming distance and BER, the distance of zero indicates that it is likely a very similar picture (or a variation of the same image). A distance that is a little bit higher than 0 means a few things may be different, but they are probably still close enough to be similar. But if the distance is closer to 0.5? That's probably a very different picture [15]. However, as was already mentioned in the Chapter 2, the maximum value of entirely different images is going to be the 1. But for image hashing task it is very unlikely that it will ever reach that value. It is possible only if the two images compared will be black and white.

Figure 18. Image comparison before and after applying Average Hash.

The most important property of this method is that its really fast and easy to implement, while at the same show quite very good results for this kind of task.

The whole process of getting the average color from the image and comparing it with the original one can be done in MATLAB in 1 line:

```
img_avg = (img >= mean2(img));
```

Figure 19. Getting binary matrix in MATLAB based on Average Hash.

In Figure 19, the "img" variable is the matrix with the luminance value of YCrCb gotten from the grayscale version of the image. As an output we will get a desired binary matrix.

To compare, how accurate the Average Hash method is, let us take a look at the same image of the "Baboon" under different small CPO attacks (Figure 8), to show the quick idea of how accurate the Average Hash is. By generating the Average Hash for each of the cases, the resulting binary hashes we get can also be represented as the images. The result of this process can be found in the Appendix as Figure 32. On the left side are the

images being processed, on the right are the image representation of the binary hashes that the Average Hash function produce. The "similarity score" is computed based on the original image (a).

For distance metrics to get the "similarity score" *Hamming distance,* and *BER* are used and compared with the original image (a). The higher Hamming distance, the more differences the image has and perceptually similar it is. Because the whole vector consists of 64 value, the maximum value for Hamming distance will be 64. That means that if the image has this metric close to 64, the image will be completely different.

The same goes to BER, which will have the same value as the *Normalized Hamming distance*. The closer BER to 0, the more perceptually similar the image is, while closer to 0.5 (technically closer to 1) defines that it is different. By varying the value of "similarity score", we can set how accurate the results are going to be.

As for comparison with entirely different images, the "similarity score" will be much higher. The Hamming distance that shows the score higher than 30 ($HD > 30$) in this case means that the images are entirely different. In the same Figure 32, on the left side is the image being processed, on the right is the image representation of the binary hash. In brackets of each example the "similarity score" is pointed out compared to the original image (a).

The whole results based on the examples provided show that the Average Hash algorithm is does a good job in image comparison. However the FPR for this method is considered to be the highest if compared to other methods [16]. This method shows not very good results in the case of rotation but is very robust to some color manipulations.

3.2.2. Difference Hash

Another very similar to Average Hash approach is the Difference Hash, and it has the same strong and weak features: excellent performance [16] while being not very accurate. The accuracy, however, strongly depends on the application scenarios it is being used. In many cases the performance outstands the need for accuracy. Even though, the Average Hash can be used for image hashing it is still not very accurate. To improve that it was evolved into the Difference Hash to make the whole process more robust to different CPO and not affect the excellent performance of the Average Hash.

Like the Average Hash, is pretty simple to implement and is far more accurate than it has any right to be [16]. Taking the same image that was produced during pre-processing step in Figure 9, but, in this case, the size that will be used during the pre-processing will be 8x9, instead of 8x8. Regardless of the image is stretched it will not affect the on "perceptual features" of the processed image. The main reason why such strange dimensions were chosen is to make the finding differences easier.

Figure 20. Pre-processed image "Baboon" before applying Difference Hash.

This size of the produced vector will be 72 bits. But at the end of the whole operation we will end up with the hash that will be only 64 bits in size. By comparing each row to the row below we will get as a result only 8 rows. That means at the end of processing we will get 64 bits of information instead of 72. To make it more clear let us have a look at the same image in Figure 20 represented as table of luminance values:

81	61	12	8	28	36	73	130
130	109	24	8	40	57	130	174
93	105	186	174	219	219	174	182
45	85	251	202	239	255	158	97
69	4	243	202	247	239	49	32
158	0	121	174	223	113	16	20
231	121	65	138	142	77	93	53
239	206	202	142	142	150	210	109
154	97	227	194	190	174	81	40

Figure 21. Luminance matrix of "Baboon" before applying Difference Hash.

Each value of each row must be compared to the values in a row below. In this case the row [81, 61, 12, 8, 28, 36, 73, 130] will be compared to [130, 109, 24, 8, 40, 57, 130, 174]. The same will go for other rows. Each elements of the processed row must be compared individually. Extending the same example, 61 will be compared to 130, 61 to

109 etc. Depending on which value is higher or lower compared to the value from the row below, a new binary matrix can be produced.

0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	0	0	0	0	1	1
0	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	0	0	1	0	0	0
1	1	0	0	0	0	1	1

Figure 22. Binary matrix of "Baboon" after applying Difference Hash.

Using the approach like in Average Hash we can get the feature vector from it:

```
0001000011000000110000110110011101111111001111000001100011000001
```

Figure 23. Feature vector after applying Difference Hash on image "Baboon".

In order to get the similarity score the same distance metrics can be used: *Hamming distance* or *BER*. The same binary matrix in can be represented as the image, where 0 is black and 1 is white. For comparison the original one is also added:

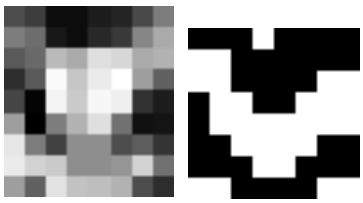


Figure 24. Example, of feature vector as image before and after Difference Hashing.

In MATLAB the desired Difference Hash function can be done using iteration:

```
result = [];
for k = 1:(size(img) - 1)
    result = [result; img(k,:) >= img(k + 1,:)];
end
```

Figure 25. Getting binary matrix in MATLAB based on Difference Hash.

In Figure 25, the "img" variable is the matrix with the luminance value of YCrCb gotten from the grayscale version of the image. As an output ("result" variable) we will get a

desired binary matrix using Difference Hash method based on the rows comparison described above.

However, the Difference Hash algorithm can be implemented not only using rows. Instead of them, the columns can be used. The only difference will be the size of the pre-processed image. If in the case of rows we have used size of 8x9, then for columns size of 9x8 needed. The only thing that must be considered is that the method you are using to find the differences must be consistent.

In order to verify the robustness property of the Difference Hash method let us take a look at the same image of the "Baboon" under different small CPO attacks (Figure 8) like we already did before when testing the Average Hash method. The Figure 33 in the Appendix also includes image representation of the resulted feature vector. The same thing goes for perceptually different images. The original image "Baboon" is compared to "Lena" and "Peppers". The result can be found in the same Figure 33.

The demonstrated examples above show that if compared to the Average Hash, the Difference Hash function seems to be much more accurate and more robust enough. The speed of both examined hash methods does not differ much, and both of them has shown the same speed in most cases. The Difference hash, however, is slightly better in terms of speed [16].

Table 6. Robustness comparison of Average Hash and Difference Hash.

	Hamming distance	
	Average Hash	***Difference Hash***
Rotated	6	6
Blurred	4	**2** ***(better)***
Color manipulations	**0** ***(better)***	1
JPEG lossy compression	2	**1** ***(better)***
Noise	3	**2** ***(better)***
Other manipulations	8	**7** ***(better)***

The results in Table 6 compare both described hashes in case of different CPO attacks. As a distance metric, the Hamming distance is used. In case of CPO, the higher the value, the less accurate method is. In total, the results state that the Difference Hash method shows less FPR if compared to the Average Hash based on the provided example. Of

course, the tests were based only on one single image (“Baboon”) and is too early to state which method is better. But in overall the Difference Hash in most cases will outstand the Average Hash in terms of accuracy and will produce less FPR. The Average Hash method, however, has shown better result when comparing image that had some color manipulation attacks, but the result, in this case, does not differ much.

3.2.3. DCT Based Hash

The Average and Difference Hashes have a simple idea behind and they are both are good for image comparison if considering the simplicity of how they work. But in most cases they are not very accurate. However, to make the feature extraction more robust a DCT Based Hash can be used.

The DCT Based Hash uses the same approach like in Average Hash: find the mean values and compare. However, it extends the average approach to the extreme, using a Discrete cosine transformation (DCT) to reduce the frequencies [15]. The main idea behind it is to get the lowest frequencies of an image. After performing a DCT, we can eliminate the coefficients that represent high frequencies that the HVS is not very sensitive to. After that, we can find the mean values of the lowest ones and compare them to the acquired mean value of them.

As you might already have guessed, the core of this feature extraction technique is DCT by itself. It is widely used in computer science, especially in image compression. For example, without DCT there wouldn’t be JPEG compression. The most amazing feature of DCT is that it later can be inverted back into the original image by using the Inversed Discrete cosine transformation (IDCT). Furthermore, it is a quite fast transform in terms of performance. For most images, much of the signal energy lies at low frequencies, which appear in the upper left corner of the DCT. However, various properties of the DCT can be utilized to create perceptual image hash functions. But the low-frequency DCT coefficients of an image are mostly stable under image manipulations [13].

The DCT uses only cosine functions while e.g. the discrete Fourier transform (DFT) uses both cosines and sines [13]. For an $N * N$ image, the DCT can be found using the following equation:

$$C(k,l) = \alpha(k,l) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i,j) \cos\left(\frac{(2i+1)k\pi}{2N}\right) \cos\left(\frac{(2j+1)l\pi}{2N}\right) \qquad (3.1)$$

where

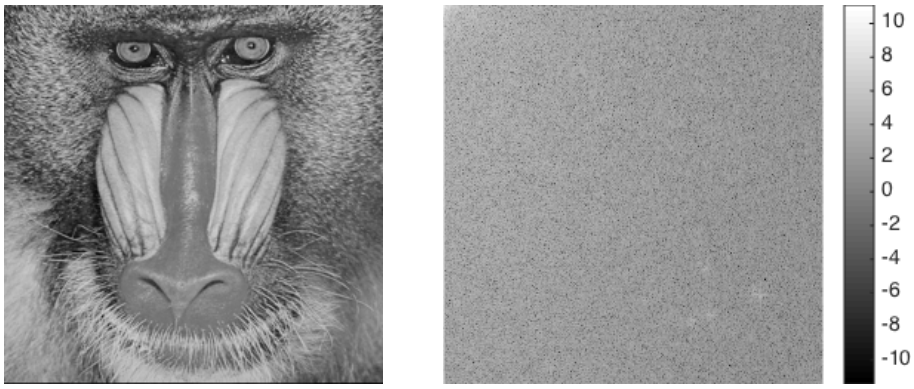
$$\alpha(k,l) = \begin{cases} \frac{1}{N}, & k,l = 0 \\ \frac{2}{N^2}, & k,l = 1,2,3,\dots,N-1 \end{cases}$$

The Equation 3.1 above is the slightly adapted DCT-II, which is the most commonly used form of it. For perceptual image hashing we are going to use the DCT-II and refer simply as DCT. If you are using MATLAB, then the DCT can be applied to image like this:

```
img = dct2(img);
```

Figure 26. Discrete Cosine Transform (DCT) in MATLAB.

In order to demonstrate, we can take the original image "Baboon" 512x512 pixels in size and get the DCT of it. The result is shown in Figure 27 below.



a. Original *b. After applying DCT*

Figure 27. Discrete Cosine Transform of the image "Baboon".

By using the IDCT we can get the same image from the DCT domain if needed. However, for perceptual image hashing that is not needed. But before using the DCT for feature extraction we must preprocess our image. For the DCT Based Hash we can used the same technique described in Chapter 3.1.1 with only one difference: we must resize the image to 32x32 instead of 8x8. This is really done to simplify the DCT computation and not because it is needed to reduce the high frequencies [15].

To verify how DCT Based Hash is robust enough to different CPO attacks (Figure 8), let us take the same pictures of the "Baboon" we have tested earlier. For each case, we will generate the fingerprint. In order to demonstrate how the acquired fingerprints will look like, we will translate it into the images. The results of this procedure can be seen in the Appendix as Figure 34.

The provided example shows that the fingerprint does not change very much from the original one in most cases. The Hamming distance shows the exact number of how many bits differ. The lower the value, the better is the algorithm. To carry on the tradition of using the same image "Baboon" let us also verify how this method will behave in case of perceptually different images by comparing it to "Lena" and "Peppers". The results can seen in the same Figure 34. In this case, the different images must have the Hamming distance (HD) close to approximately 32 or even higher in some cases. The same thing goes with Bit error rate (BER), the value must be approximately 0.5 or higher for perceptually different images. The "similarity scores" for the examples in Figure 34 show that the DCT based hash has successfully found different images.

Of course, we cannot state which algorithm is better based only on one example. But in order to give you the idea of which is more accurate based on provided example, let us compare DCT Based Hash to Difference Hash. The Difference Hash has shown much better results for images under typical CPO attacks if comparing to the Average hash, so in the Table 7 we will compare it to DCT Based hash.

Table 7. Robustness comparison of Difference Hash and DCT Based Hash.

	Hamming distance	
	Difference Hash	***DCT Based Hash***
Rotated	**6** ***(better)***	7
Blurred	2	2
Color manipulations	1	1
JPEG lossy compression	**1**	**0** ***(better)***
Noise	**2**	**0** ***(better)***
Other manipulations	**7**	**4** ***(better)***

In total, the results in Table 7 state that the DCT Based Hash method show less FPR if compared to the Difference Hash and the same thing goes with Average Hash. For the

provided example DCT Based hash is much more robust under different CPO attacks that other described before algorithms. It has shown the same or much better results in almost all cases. The strongest feature of the DCT Based hash is that it is robust to noise and JPEG lossy compression. However, in case of rotation the DCT Based Hash has shown less accurate score. In the case of different images, all hashes have successfully succeeded in the task and have shown the Hamming distance higher than 30. The small fluctuations in case of completely different images do not give much information on which algorithm is better. However, if considering the Hamming distance, then the Difference Hash have shown slightly better results for finding different images.

3.2.4. Other algorithms

The described above algorithms are considered to be the most basic ones used in perceptual image hashing, but they give you a quite good idea of how perceptual hashing works. Furthermore, they show very impressive results even thought they seem to be quite simple. Most of the existing image hashing studies mainly focus on the feature extraction stage and use them during authentication, which can roughly be classified into the four following categories [19]:

- ***Statistic-based schemes:*** This group of schemes extracts hash features by calculating the images statistics such as mean, variance, higher moments, etc.
- ***Relation-based schemes:*** This category of approaches generates the hash based on features by making use of some invariant relationships of the coefficients of Discrete Cosine Transform (DCT) or Wavelet Transform (DWT).
- ***Coarse-representation-based schemes:*** In this category of methods, the perceptual hashes are calculated by making use of coarse information of the whole image, such as the spatial distribution of significant wavelet coefficients, the low-frequency coefficients of Fourier transform, and so on.
- ***Low-level feature-based schemes:*** The hashes are extracted by detecting the salient image feature points. These methods first perform the DCT or DWT transform on the original image, and then directly make use of the coefficients to generate final hash values. However, these hash values are very sensitive to global as well as local distortions that do not cause perceptually significant changes to the images.

However, there also other ways of classification the PHF's, but this classification seems to be much more accurate. Each of the proposed schemes have different approaches of perceptual image hashing, and this work cannot describe all of them and give examples. Some of them are more complex and give better results and less FPR, others take less calculation time but the results are of testing the robustness show that it can be improved. But all of them use can use the same perceptual image hashing framework. It is clear that designing an image authentication scheme that offers high robustness and tamper detection capability while being secure like a cryptographic hash function is a very challenging task.

For example, for *"Statistic-based schemes"* a compact hash is proposed by Lou and Liu in [22]. They divide the image into non-overlapping blocks of 8x8 pixels and calculate the mean value of each block. The mean value is then quantized to get the image hash. A 2-bit code is used to get four-level intensity quantization of the mean value of each block. For an image of size 256x256 pixels, the hash size reported is 2048 bits, tolerance to compression and to tamper localization capability [21].

In [23], a good example for "Relation-based schemes" was proposed Zhao. His image hashing scheme based on features obtained from color histogram. The image is first converted from RGB color space to HSI color space as HSI is more suitable to analyze the color perception features then the RGB model. Each component of HSI is then quantified to obtain a one-dimension vector that is further normalized to 24 elements and histogram of these elements is then obtained. The binary sequence obtained from the histogram is permuted using a secret key to obtain the final hash [21].

In [24], Monga and Evans proposed a very interesting way of getting the hash based on "*Coarse-representation-based schemes"* by using visually significant feature points. A wavelet-based iterative feature detection algorithm is used to extract these feature points. A feature vector is generated based on the extracted feature points. At first the input image is divided into overlapping circular/elliptical regions with randomly selected radii. These regions are then approximated as rectangles using water-filling like approach. The deterministic algorithm is then applied to the random rectangles to get the binary hash value [21].

A fascinating example of “Low-level feature-based schemes” was presented by Tang in [25]. The proposed by him, method also has three stages. In the first stage, preprocessing of an input image is done in the way we are using in out work. In the feature extraction step, ring division is done by calculating the circle radius and the distance from each pixel to the image center. It is observed that the image contents of the original image’s ring were unchanged after rotation. In the final stage, the entropy of each ring is calculated to generate the image hash. The number of rings represents the length of the hash function [21].

3.3. Summary

The perceptual image hashing must address should address three important issues: robustness to non-malicious manipulations, ability to detect malicious tampering with localization capability and security [21]. Interestingly, all these issues are related to each other. For example, increasing robustness to non-malicious manipulations reduces the tamper detection capability and security of the overall scheme. Similarly, if the system is made very sensitive to detect malicious tampering, the robustness parameter will suffer [21].

In this chapter, we have tried to propose the framework that can be used in perceptual image hashing by generalizing the whole process of obtaining the hash from the image and comparing it with the database. The core of the framework, however, is the hash generation that consists of only three steps: pre-processing, feature extraction and post-processing. The pre-processing step usually is needed only to prepare the image for the feature extraction and to standardize the image for easier calculations. Even though, the pre-processing step also covers some of the basic techniques to enchase the robustness qualities, the main purpose of it is the standardization. The robustness property of image hashing arises from robust feature extraction. The post-processing mainly is needed only for final hash generation to quantize the extracted feature vector. Furthermore, it also can be used to enhance the security of the final hash by using traditional cryptographic hashing techniques if needed. However, in most cases this is unneeded step and strongly depends on the ways of which algorithms are used.

In addition to the framework and hash generation schemes, different feature extraction techniques also known as the perceptual image hash functions were proposed and

described. Each defined hash method has its advantages and accuracy threshold. However, in most cases based on the evaluation of the results of how each method behaves under different CPO attacks and in case of different images, the DCT Based Hash has shown the best results if comparing to the Average and Difference Hash methods. For evaluation process the BER and Hamming distance were used and results were provided in the comparison table when describing each method. Furthermore, other approaches of perceptual image hash functions were mentioned and were generalized into four main groups.

4. Implementation in MATLAB

The previous chapter discussed and reviewed the theoretical background behind some of the perceptual image hash functions and also the proposed framework. This chapter covers the actual implementation of the described perceptual hash functions in MATLAB: Average, Difference and DCT Based. The implementation considers using the proposed hash generation scheme. However, the post-processing step is intentionally omitted, because the described algorithms do not cover post-processing operations in terms of security based on traditional cryptographic hash methods. Because, in most cases, it will make the hash irreversible. We will not be able to compare distances between two irreversible hashes in the current solution. However, the implementation covers the security aspect as well by implementing the secret key usage. Furthermore, the proposed perceptual image hash algorithms does not need any additional quantization steps because they all produce the same fixed size binary hash equal to 64 bits. For current implementation the both feature extraction and post-processing steps can be generalized into one method: *process()*.

The main program that is being used throughout the whole work is MATLAB, which is the high-level language and interactive environment that makes the signal and image processing much easier. In some of the previous chapters some basic code snippets in MATLAB were already introduced, however, the final implementation is described in this chapter and the code is provided in the Appendix.

4.1. Structure

The whole implementation of each described earlier hash function and distance metrics are divided into separated classes which are defined in MATLAB as “classdef” blocks. Each so called “classes” are completely separate from each other so no connections between them are actually needed. The whole project implementation structure can be divided into four different classes that will hold all the logic. In addition to that, several extra scripts are also provided in the Appendix that simply test out the use of those classes. Also an additional class Image is provided in the Appendix, as the helper for image loading and magnification used in the testing scripts. However, it does not hold any logical purpose for the whole implementation was created only to simplify the testing scripts creation.

4.2. Distance/similarity metrics

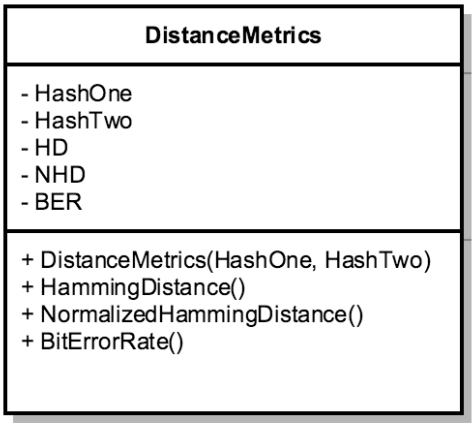


Figure 28. DistanceMetrics class.

The DistanceMetrics class plays a very important role to calculate the distance between two hashes. Because the final hashes are binary, the Euclidian distance is omitted, however it can be easily added as the new method and the code for it was already provided before in Figure 4.

The class implements different distance/similarity metrics for perceptual hashing based on the theoretical discussion earlier in Chapter 2. The code of calculation each distance was already introduced in the description of those methods and all use the internal MATLAB functions. The class by itself has a constructor that requires two parameters: first and second hashes to compare which are represented as the binary vectors. By calling the desired method we can the distance value between both hashes.

4.3. Perceptual image hash functions

Each of the described perceptual hash function method consists of the same parameters and methods. In MATLAB the access level of parameters can be configured, but in our case all are represented as public to ease the debugging. The methods of each class represent the perceptual hashing framework hash generation scheme. However, the post-processing step was intentionally omitted as was already mentioned before.

The method *process()* is using the secret key in perceptual image hash function class to generate the vectors. It using control random number generation using the internal MATLAB function *rng()* to get the same random numbers based on the key which

represents the seed. After image feature extraction the produced binary matrix is shuffled using the internal methods *randperm()* and *reshape()* to get the desired effect. Of course, the *rng()* function is firstly applied with the same key before getting the random numbers. By varying the key value we can produce completely random binary hashes that still can be compared with the hashes produced using the same method if know the key.

4.3.1. Average Hash

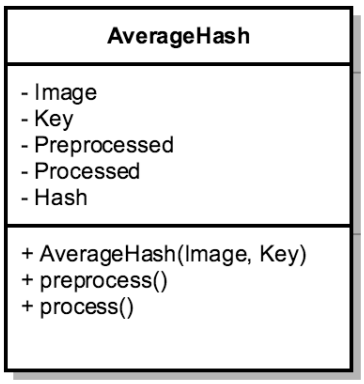


Figure 29. AverageHash class.

The *AverageHash* class (Figure 29) implements the Average perceptual image hash function, which was theoretically discussed earlier in Chapter 3.2.1. In pre-processing step which is represented as the *preprocess()* function is doing the following procedure:

1. Blurring the original image.
2. Resizing it to 8x8 using bicubic interpolation.
3. Converting it into grayscale if needed.
4. Normalizing the color using the internal *histeq()* function.

The resulted matrix 64 bits in size is saved into the local class variable *Preprocessed*. After calling the *process()* function the Average Hash algorithm is applied and the new binary matrix is shuffled using the same control random number generation method and saved to local class variable *Processed.* As a final measure the binary vector is produced from the matrix and save to variable *Hash*.

To enhance the security even more, the Average hash can also use the same reshuffling method in the *preprocess()*. The mean value will not change and it will not affect on comparison results but the resulting vector will be more secure.

4.3.2. Difference Hash

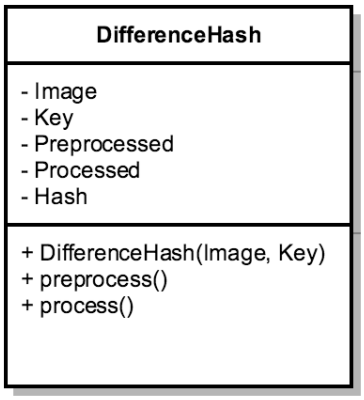


Figure 30. DifferenceHash class.

The *DifferenceHash* class (Figure 30) implements the Difference perceptual image hash function, which was theoretically discussed earlier in Chapter 3.2.2. In pre-processing step which is represented as the *preprocess()* function is doing the following procedure:

1. Blurring the original image.
2. Resizing it to 9x8 using bicubic interpolation.
3. Converting it into grayscale if needed.
4. Normalizing the color using the internal *histeq()* function.

The resulted matrix 72 bits in size is saved into the local class variable *Preprocessed*. After calling the *process()* function the Difference Hash algorithm is applied and the new binary matrix with 64 bits in size is shuffled using the control random number generation method and saved to local class variable *Processed.* Then the binary matrix is converted into the binary vector and saved as the hash into the *Hash* variable.

4.3.3. DCT Based Hash

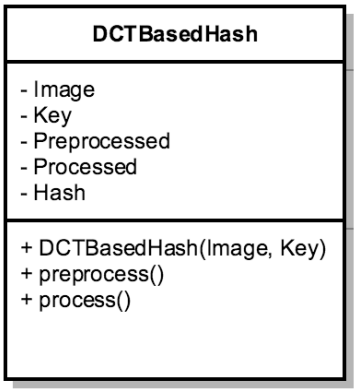


Figure 31. DCTBasedHash class.

The *DCTBasedHash* class (Figure 31) implements the DCT Based perceptual image hash function, which was theoretically discussed earlier in Chapter 3.2.3. In pre-processing step which is also represented as the *preprocess()* function the image is being prepared for the feature extraction process:

1. Blurring the original image.
2. Resizing it to 32x32 using bicubic interpolation.
3. Converting it into grayscale if needed.
4. Normalizing the color using the internal *histeq()* function.

The resulted matrix 1024 bits in size is saved into the local class variable *Preprocessed*. After calling the *process()* function the DCT is applied and the lowest frequencies are extracted from the top left corner (8x8). The resulted 64 bit matrix is then converted into the vector and the mean value is calculated omitting the first value to eliminate complete flat surfaces (colors). By using the same approach as the Average Hash, each value is compared to the mean value to quantize the data. The resulted 64 bits in size matrix is shuffled using the control random number generation method and saved to local class variable *Processed.*

Summary

In this thesis, the goals are pointed out to find the differences between cryptographic and perceptual hash functions, study different distance/similarity metrics, propose the image hashing framework and investigate different perceptual image hash functions.

The author has reviewed both cryptographic and perceptual hash functions. Furthermore, the author has defined three main properties of cryptographic and four for perceptual hash functions. For both of them were given different application scenarios and also were mentioned the main differences between them. Each defined property was briefly described, and some examples were proposed.

In addition, the author has also investigated different distance/similarity metrics that can be used in perceptual image hashing to evaluate the robustness and discriminative capabilities of image hashing schemes and has provided the ways how they can be calculated. In total four metrics were proposed and at least two of them were used in experimental results for evaluation.

Furthermore, the author has proposed the perceptual image hashing framework that consists of all required steps to produce the final hash. Each step is described, and some examples were given to explain the main purpose. In addition, three basic perceptual image hash functions that can be used during the feature extraction were proposed and analyzed. For each method were given examples of how robust each of the described methods is by analyzing the same image under different content processing operations. The final results of the robustness and discriminative capabilities of the described methods were compared, and the most accurate of the proposed methods was found.

In the end, the author describes an implementation of the perceptual image hashing in MATLAB using the theoretically discussed earlier distance/similarity metrics and three perceptual image hash functions. The final code of the whole implementation the author has added in the Appendix and included the script used for evaluation.

Kokkuvõte

Antud lõputöös on esitatud eesmärgid, et leida erinevuse krüptograafiliste ja pertseptuaalse räsimise funktsioonide vahel, õppida erinevaid kauguse/sarnasuse meetrikaid, pakkuda piltide räsimise raamistikke ja uurida piltide pertseptuaalse räsimise funktsioone.

Autor on vaadanud nii krüptograafilisi kui ka pertseptuaalse räsimise funktsioone. Samuti on autor defineerinud krüptograafia kolm põhilist omadust ja pertseptuaalse räsimise neli funktsiooni. Mõlema jaoks oli antud erinevad rakendamise stsenaariume ja nendevahelised põhilised erinevused olid samuti mainitud. Iga defineeritud omadus oli lühidalt kirjeldatud ja mõned näited olid esile toodud.

Lisaks on autor uurinud erinevaid kauguse/sarnasuse meetrikaid, mida on võimalik kasutada piltide pertseptuaalsel räsimisel, mida on võimalik kasutada jõulisuse ja piltide räsimisskeemade diskrimineerivuse võime hindamiseks, ja on esitanud võimalusi, kuidas neid on võimalik arvutada. Kokku oli esitatud neli meetrikat ja vähemalt kaks nendest olid kasutatud eksperimentaalsete tulemuste hindamiseks.

Samuti on autor pakkunud piltide pertseptuaalse räsimise raamistiku, mis koosneb kõikidest vajalikkudest sammudest, et tekitada lõplikut räsi. Iga samm on kirjeldatud ja mõned näited on antud, et seletada põhilisi funktsioone. Lisaks autor pakub kolm põhilist piltide pertseptuaalse räsimise funktsioone, mida on võimalik kasutada omaduste ekstraheerimise jooksul, olid esitatud ja analüüsitud. Iga meetodi jaoks olid antud mõned näited ja uuritud, kui jõuline on iga kirjeldatud meetod, analüüsides sama pilti erinevate sisu töötlemise operatsioonide rakendamisel. Kirjeldatud meetodite jõulikusse ja diskrimineerivuse võimsuse lõplikud tulemused olid võrreldud ja kõige täpsem pakutud meetoditest oli leitud.

Antud töö lõpuks kirjeldab autor piltide pertseptuaalse räsimise implementatsiooni MATLAB-s kasutades varem teoreetiliselt arutatud kauguse/sarnasuse meetrikaid ja kolm piltide pertseptuaalse räsimise funktsioone. Tervikliku implemetatsiooni lõpliku koodi on autor lisanud lisasse ning samuti lisas hindamiseks kasutatavat skripti.

References

[1] Schneier, Bruce. “Cryptanalysis of MD5 and SHA: Time for a New Standard”. *Computerworld*, August 2004.

[2] Qin, Chuan, Chin-Chen Chang, and Pei-Ling Tsou. “Perceptual Image Hashing Based on the Error Diffusion Halftone Mechanism”. *International Journal of Innovative Computing, Information and Control* (September 2012).

[3] Black, Paul. “Hamming distance”. 31 May 2006. http://xlinux.nist.gov/dads/HTML/HammingDistance.html (accessed 5 May 2015).

[4] Hamming, Richard W. “Error detecting and error correcting codes.” *Bell System Technical*, April 1950.

[5] Rajarajeswari, P., и N. Uma. “A Study of Normalized Geometric and Normalized Hamming Distance Measures in Intuitionistic Fuzzy Multi Sets.” *International Journal of Science and Research (IJSR)* (November 2013).

[6] Swaminathan, Ashwin, Yinian Mao, and Min Wu. “Robust and Secure Image Hashing.” *IEEE Transactions on Information Forensics and Security* (June 2006).

[7] Wu, Min, and Bede Liu. “Multimedia Data Hiding”. New York: Springer-Verlag, 2002.

[8] Cox, I. J., M. L. Miller, и J. A. Bloom. “Digital Watermarking”. San Mateo, CA: *Morgan Kaufmann*, 2001.

[9] Menezes, Alfred J., Paul C. Oorschot, и Scott A. Vanstone. *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 1998.

[10] Wang, Yong. “New Way to Construct Cryptographic Hash Function”. 16 February 2014.

[11] Venkatesan R, Koon S-M, Jakubowski MH, Moulin P. Robust image hashing. IEEE Int. Conf. on Image Process. 2000; pp. 664–6.

[12] Hernandez, Ricardo Antoio Parrao, and Brian M. Kurkoski. “Robust Image Hashing Using Image Normalization and SVD Decomposition”. *IEEE*, 10 August 2011.

[13] Zauner, Christoph. “Implementation and Benchmarking of Perceptual Image Hash Functions”. *Self-publisher*. 2010.

[14] Kandepet, Deepak. “Detecting Similar and Identical Images Using Perseptual Hashes”. 30 July 2012. http://hackerlabs.org/blog/2012/07/30/organizing-photos-with-duplicate-and-similarity-checking/ (accessed: 1 May 2015).

[15] Krawetz, Neal. “Looks Like It”. 26 May 2011. http://www.hackerfactor.com/blog/?/archives/432-Looks-Like-It.html (accessed: 1 May 2015).

[16] Krawetz, Neal. “Kind of Like That”. 21 January 2013. http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html (accessed: 1 May 2015).

[17] Leskovec, Jure, Anand Rajaraman, and Jeffrey D. Ullman. “Mining of Massive Datasets”. *Stanford: Stanford University*, 2010.

[18] Cabeen, Ken, и Peter Gent. “Image Compression and the Discrete Cosine Transform." *Math 45*. College of the Redwoods.

[19] Azhar Hadmi, William Puech, Brahim Ait Es Said and Abdellah Ait Ouahman (2012). “Perceptual Image Hashing, Watermarking - Volume 2”, Dr. Mithun Das Gupta (Ed.), ISBN: 978-953-51-0619-7, InTech, DOI: 10.5772/37435.

[20] Lu, Chun-Shien, and Hong-Yuan Mark Liao. “Structural Digital Signature for Image Authentication: An Incidental Distortion Resistant Scheme”. *IEEE Transactions on Multimedia* 5 (2003).

[21] Zivic, Natasa. “Robust Image Authentication in the Presence of Noise”. Natasa Zivic (Ed.). *Siegen: Springer*, 2015.

[22] Lou D-C, Liu J-L. “Fault resilient and compression tolerant signature for image authentication”. *IEEE Trans Consumer Electron.* 2000;46(1):31–9.

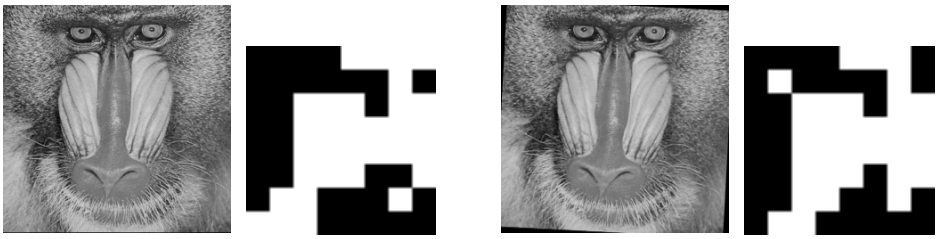
[23] Zhao Y, Gu C, Wei W. “Image hashing based on color histogram”. *J Inf Comput Sci.* 2012;9(15):4397–404.

[24] Monga V, Evans BL. “Perceptual image hashing via feature points: performance evaluation and tradeoffs”. *IEEE Trans Image Process.* 2006;15(11):3452–65.

[25] Tang Z, Zhang X, Huang L, Dai Y. “Robust image hashing using ring-based entropies”. *Signal Process.* 2013;93:2061–9.

[26] Preneel, Bart. “Cryptographic hash functions: an overview”. *ICSVC* (1993)

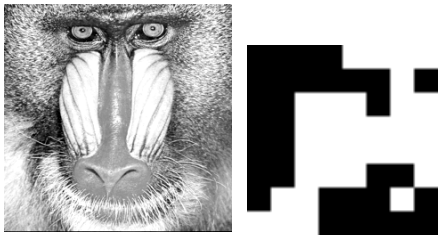
Appendix 1 – Additional Figures



a. Original

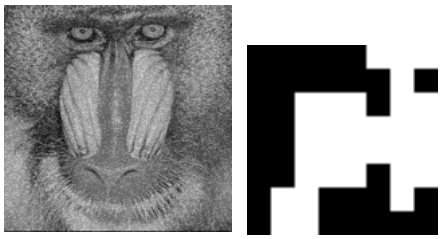
b. Rotated (BER: 0.094, HD: 6)

c. Blurred (BER: 0.063, HD: 4)



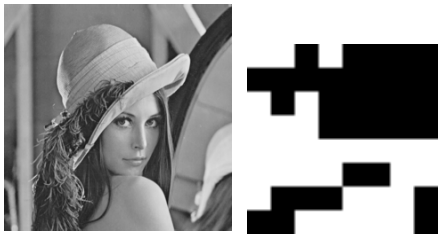
d. Color manipulations (BER: 0, HD: 0)

e. JPEG lossy compression (BER: 0.031, HD: 2)



f. Noise (BER: 0.047, HD: 3)

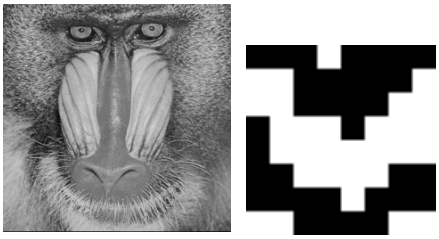
g. Other manipulations (BER: 0.13, HD: 8)



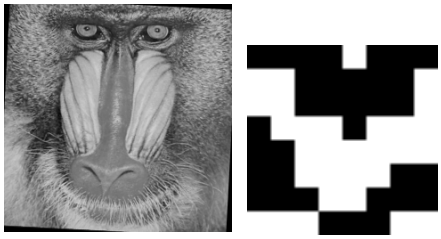
h. "Lena" (BER: 0.58, HD: 37)

i. "Peppers" (BER: 0.5, HD: 32)

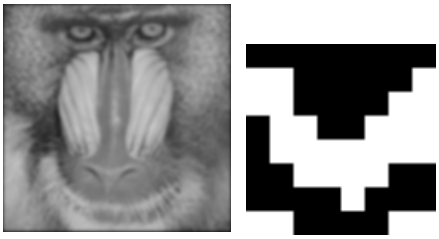
Figure 32. Average Hash behavior in case different images and CPO.



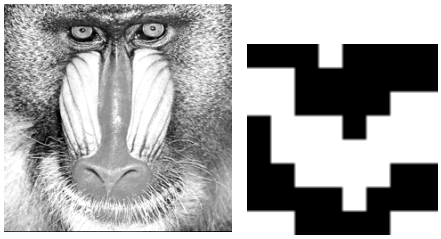
a. Original



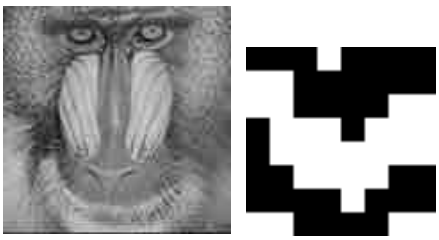
b. Rotated (BER: 0.09, HD: 6)



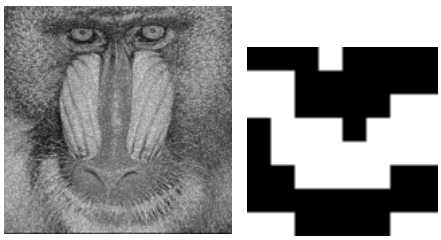
c. Blurred (BER: 0.03, HD: 2)



d. Color manipulations (BER: 0.02, HD: 1)



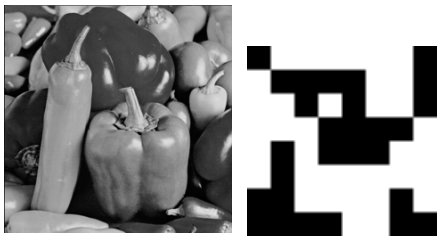
e. JPEG lossy compression (BER: 0.02, HD: 1)



f. Noise (BER: 0.03, HD: 2)

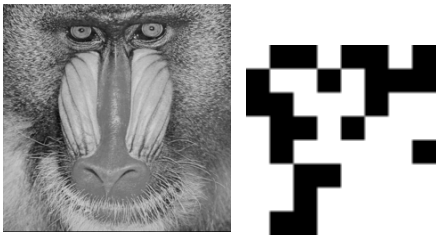
g. Other manipulations (BER: 0.11, HD: 7)

h. "Lena" (BER: 0.56, HD: 36)

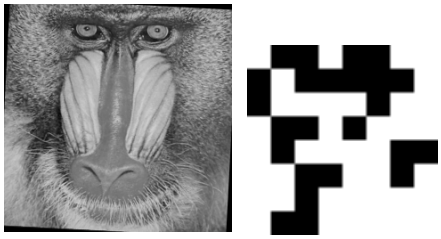


i. "Peppers" (BER: 0. 53, HD: 34)

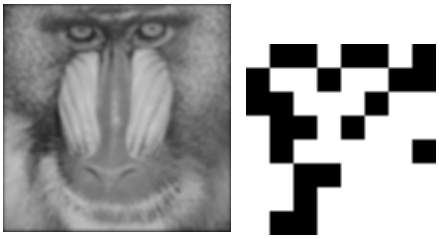
Figure 33. Difference Hash behavior in case different images and CPO.



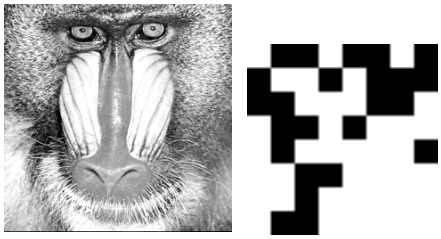
a. Original



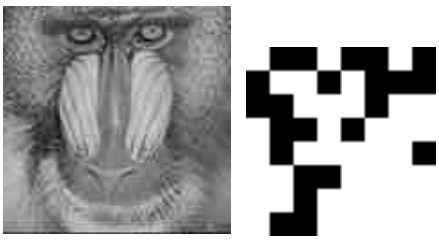
b. Rotated (BER: 0.11, HD: 7)



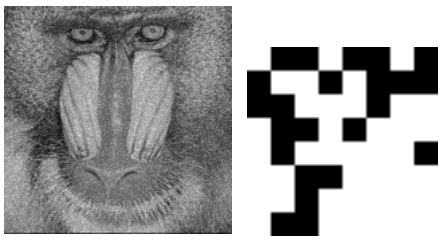
c. Blurred (BER: 0.03, HD: 2)



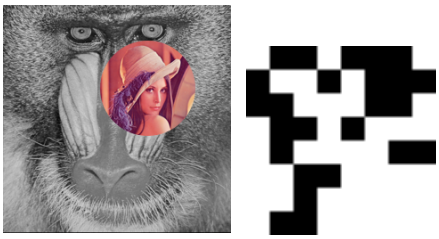
d. Color manipulations (BER: 0.02, HD: 1)



e. JPEG lossy compression (BER: 0, HD: 0)

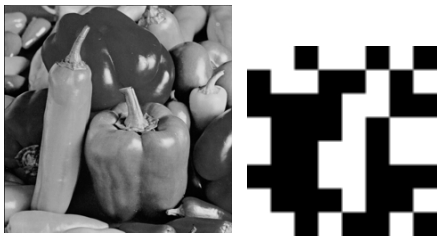


f. Noise (BER: 0, HD: 0)



g. Other manipulations (BER: 0.06, HD: 4)

h. "Lena" (BER: 0.52, HD: 33)



i. "Peppers" (BER: 0.46875, HD: 30)

Figure 34. DCT Based Hash behavior in case different images and CPO.

Appendix 2 – Code in MATLAB

The Average Hash function implementation described in Chapter 4.3.1 and theoretically discussed in Chapter 3.2.1:

```
classdef AverageHash

    % AVERAGEHASH  Performs hash generation using Average Hash
technique.
    %
    % Description
    % RESULT = AverageHash(IMG);
    %   IMG    - image to process
    %   RESULT - AverageHash object

    % Authored May 2015 by Victor Popkov

    properties
        Image
        Key
        Preprocessed
        Processed = []
        Hash
    end

    methods
        function obj = AverageHash(Image, Key)
            % class constructor
            if (nargin > 0)
                obj.Image = Image;
                obj.Key = Key;
            end
        end

        function obj = preprocess(obj)
            % pre-processing

            % Blurring
            filter = fspecial('gaussian', [3 3], 2);
            obj.Preprocessed = imfilter(obj.Image, filter, 'same');
            % Resizing
            obj.Preprocessed = imresize(obj.Preprocessed, [8 NaN],
'bicubic');
            % Color reduction
            [rows, columns, channels] = size(obj.Preprocessed);
            if channels > 1
                obj.Preprocessed = rgb2gray(obj.Preprocessed);
            end
            % Color normalizations
            obj.Preprocessed = histeq(obj.Preprocessed);

            % Security (shuffle)
            if obj.Key ~= 0
                rng(obj.Key, 'twister');
                rng(obj.Key + randi([1, numel(obj.Preprocessed)]),
'twister');
                new_order = randperm(numel(obj.Preprocessed));
                obj.Preprocessed =
```

```
reshape(obj.Preprocessed(new_order), size(obj.Preprocessed));
            end
        end

        function obj = process(obj)
            % processing

            % Getting feature matrix
            obj.Processed = (obj.Preprocessed >=
mean2(obj.Preprocessed));

            % Security (shuffle)
            if obj.Key ~= 0
                rng(obj.Key, 'twister');
                rng(obj.Key + randi([1, numel(obj.Preprocessed)]),
'twister');
                new_order = randperm(numel(obj.Processed));
                obj.Processed = reshape(obj.Processed(new_order),
size(obj.Processed));
            end

            obj.Hash = double(obj.Processed(:)');
        end
    end

end
```

The Difference Hash function implementation described in Chapter 4.3.2 and theoretically discussed in Chapter 3.2.2:

```
classdef DifferenceHash

    % DIFFERENCEHASH  Performs hash generation using Difference Hash
technique.
    %
    % Description
    % RESULT = DifferenceHash(IMG);
    %   IMG    - image to process
    %   RESULT - DifferenceHash object

    % Authored May 2015 by Victor Popkov

    properties
        Image
        Key
        Preprocessed
        Processed = []
        Postprocessed
        Hash
    end

    methods
        function obj = DifferenceHash(Image, Key)
            % class constructor
            if (nargin > 0)
                obj.Image = Image;
```

```
                obj.Key = Key;
            end
        end

        function obj = preprocess(obj)
            % pre-processing

            % Blurring
            filter = fspecial('gaussian', [3 3], 2);
            obj.Preprocessed = imfilter(obj.Image, filter, 'same');
            % Resizing
            obj.Preprocessed = imresize(obj.Preprocessed, [9 8],
'bicubic');
            % Color reduction
            [rows, columns, channels] = size(obj.Preprocessed);
            if channels > 1
                obj.Preprocessed = rgb2gray(obj.Preprocessed);
            end
            % Color normalizations
            obj.Preprocessed = histeq(obj.Preprocessed);
        end

        function obj = process(obj)
            % processing

            % Difference hash
            for k = 1:(size(obj.Preprocessed) - 1)
                obj.Processed = [obj.Processed; obj.Preprocessed(k,:)
>= obj.Preprocessed(k + 1,:)];
            end

            % Security (shuffle)
            if obj.Key ~= 0
                rng(obj.Key, 'twister');
                new_order = randperm(numel(obj.Processed));
                obj.Processed = reshape(obj.Processed(new_order),
size(obj.Processed));
            end

            obj.Hash = double(obj.Processed(:)');
        end
    end

end
```

The Difference Hash function implementation described in Chapter 0 and theoretically discussed in Chapter 3.2.3:

```
classdef DCTBasedHash

    % DCTBasedHash  Performs hash generation using Discrete cosine
transform.
    %
    % Description
    % RESULT = DCTBasedHash(IMG);
    %   IMG    - image to process
    %   RESULT - DCTBasedHash object
```

```
    % Authored May 2015 by Victor Popkov

    properties
        Image
        Key
        Preprocessed
        Processed = []
        Postprocessed
        Hash
    end

    methods
        function obj = DCTBasedHash(Image, Key)
            % class constructor
            if (nargin > 0)
                obj.Image = Image;
                obj.Key = Key;
            end
        end

        function obj = preprocess(obj)
            % pre-processing

            % Blurring
            filter = fspecial('gaussian', [3 3], 2);
            obj.Preprocessed = imfilter(obj.Image, filter, 'same');
            % Resizing
            obj.Preprocessed = imresize(obj.Preprocessed, [32 32],
'bicubic');
            % Color reduction if required
            [rows, columns, channels] = size(obj.Preprocessed);
            if channels > 1
                obj.Preprocessed = rgb2gray(obj.Preprocessed);
            end
            % Color normalizations
            obj.Preprocessed = histeq(obj.Preprocessed);
        end

        function obj = process(obj)
            % processing

            img = obj.Preprocessed;

            img = dct2(img, [32 32]);

            img = img(1:8, 1:8);

            vector = img(:)';
            vector = vector(2:length(vector));
            mean = mean2(vector);

            img = (img >= mean);

            obj.Processed = img;

            % Security (shuffle)
            if obj.Key ~= 0
                rng(obj.Key, 'twister');
                new_order = randperm(numel(obj.Processed));
```

```
                obj.Processed = reshape(obj.Processed(new_order),
size(obj.Processed));
            end

            obj.Hash = double(obj.Processed(:)');
        end
    end

end
```

The Distance/Similarity metrics implementation theoretically discussed in Chapter 2:

```
classdef DistanceMetrics

    % DISTANCEMETRICS  Performs distance/similarity calculation for
    %                  two hashes. Available:
    %                    - Hamming distance (HD)
    %                    - Normalized Hamming distance (NHD)
    %                    - Bit error rate (BER)
    %
    % Description
    % RESULT = DistanceMetrics(HASHONE, HASHTWO);
    %   HASHONE - the main hash
    %   HASHTWO - the second hash to compare with first one
    %   RESULT  - DistanceMetrics object

    % Authored May 2015 by Victor Popkov

    properties
        HashOne
        HashTwo
        HD
        NHD
        BER
    end

    methods
        function obj = DistanceMetrics(HashOne, HashTwo)
            % class constructor

            if (nargin > 0)
                obj.HashOne = HashOne;
                obj.HashTwo = HashTwo;
            end
        end

        function obj = HammingDistance(obj)
            % Hamming distance (HD)

            obj.HD = pdist2(obj.HashOne, obj.HashTwo, 'hamming') *
length(obj.HashOne);
        end

        function obj = NormalizedHammingDistance(obj)
            % Normalized Hamming distance (NHD)

            obj.NHD = pdist2(obj.HashOne, obj.HashTwo, 'hamming');
```

```
        end

        function obj = BitErrorRate(obj)
            % Bit error rate (NHD)

            obj.BER = biterr(obj.HashOne, obj.HashTwo) /
length(obj.HashOne);
        end
    end

end
```

Code used to generate the hashes and calculate the distance for evaluation purposes. Hashed are represented as the images and are magnified x8:

```
source   = 'examples/thesis/images';
dest     = 'examples/thesis/hashed/average'; % average, difference,
dct
dest_ext = '@8x.png';

algorithm = @DCTBasedHash; % Choose algorithm to use (AverageHash,
DifferenceHash, DCTBasedHash)

key = 175; % if equals 0, then no security

images = dir(fullfile(source, '*'));

% Original image hash
o = algorithm(imread(fullfile(source, 'original.png')), key);
o = o.preprocess();
o = o.process();

for k = 3:numel(images)
    img = imread(fullfile(source, images(k).name));

    [pathstr, name, ext] = fileparts(images(k).name);

    % Hashing algorithm
    f = algorithm(img, key);
    f = f.preprocess();
    f = f.process();

    file = Image(fullfile(pathstr, [ name dest_ext ]), dest);
    file = file.setContent(f.Processed);
    file.magnify(64).save();

    d = DistanceMetrics(o.Hash, f.Hash);
    d = d.NormalizedHammingDistance();
    d = d.HammingDistance();
    d = d.BitErrorRate();

    fprintf('   %s:\n   %s, BER: %s, HD: %s, NHD: %s\n\n',
images(k).name, num2str(f.Hash, '%1d'), num2str(d.BER), num2str(d.HD),
num2str(d.NHD));
end
```

```
clearvars images o k img pathstr name ext f d file;
```

Image class used to simplify the process of image loading and magnification:

```
classdef Image

    properties
        Directory
        Filename
        Content
    end

    methods
        function obj = Image(Filename, Directory)
            % class constructor
            if(nargin > 0)
                obj.Filename  = Filename;
                obj.Directory = Directory;
            end
        end

        function obj = setContent(obj, content)
            obj.Content = content;
        end

        function obj = magnify(obj, size)
            obj.Content = imresize(obj.Content, [size NaN],
'nearest');
        end

        function save(obj)
            imwrite(obj.Content, fullfile(obj.Directory,
obj.Filename));
        end
    end

end
```