# Blockhashing as a forensic method

**Kurt H. Hansen**

A minor thesis submitted in part fulfilment of the degree of M.Sc. in Forensic Computing and Cyber Crime Investigation with the supervision of Dr. Nhien An Le Khac

School of Computer Science and Informatics

University College Dublin

August 1, 2016

# Abstract

In computer forensics investigation, there has always been a battle in which the offenders find new methods to hide their illegal activity and the investigator find countermeasures to these methods.

The most common method to use to hide illegal activity is to hide data connected to the illegal activity by making the material unavailable. There are several methods to make data less available. These could be techniques to encrypt the content, to hide the content by using steganography or just erase the compromising files. Erasing data content is probably the most common method to get rid of compromised data. There are several techniques to erase data files, but the most common is to use a file explorer in the operating system to erase the file. Such erasure does not have any impact on the actual data content, only the file meta-data. More sophisticated tools both erase the file meta-data and overwrite the file content with other more or less random content.

The most common method, using the file explorer to remove the file from the file listing is a prerequisite for this project. We call this ordinary file erasure. Files erased this way will have the content unchanged in an unpredictable time of period, but as the time goes, more more of the erased content and will be overwritten by new files.

There are already methods to reveal file content erased by ordinary file erasure. These methods include file carving that searches for patterns to make it possible to reveal the content. File carving is a method if the erased file content is not overwritten, but as the file content is increasingly overwritten, the file carving method is less relevant. When files are partially overwritten, there are still possibilities to identify the original content from the existing fragments.

Technically, it is possible to identify pieces of information compared to other reference files and research papers have proved this by comparing small pieces of data from a file system with pieces of data from reference material. The technique is known, but the problem of implementing this as a forensic method in an investigation has not yet been solved so far.

In previous work, the technique is demonstrated in relatively small amount of data and there is no research to implement this as a valid method that ensure the findings can be used as admissible evidence in court.

The contribution of this work is to conduct a research by using larger datasets and evaluate block hashing as a forensic valid method. The goal of the proposed project is to describe a robust methodology to use block-hashing as a forensic method to discover fragments of previously stored objects.

# Acknowledgement

I would like to express my gratitude to my supervisor Dr. Nhien An Le Khac for the useful comments, remarks and engagement throughout the learning process of this master thesis.

Furthermore I would like to thank Dr. Fergus Toolan at the Norwegian Police University (NPUC) for valuable input during the process of writing the thesis and proofreading this document.

Also, i would like to thank Associate Professor Rune Nordvik at NPUC, who has willingly shared his precious time during the process of writing the thesis and have been a valuable sparring partner during the process.

I would also like to express my gratitude to my employer NPUC that offered me time to do a lot of study during my working hours with a special thank you to Police Inspector Ivar L. K. Husby at the NPUC.

# Contents

**I Introduction 10**

**1 Introduction 11**
1.1 Motivation and Background . . . . . 11

**2 Litterature Survey 13**
2.1 Forensic Methodology and Legal aspects . . . . . 13
2.2 Hash algorithms, Entropy and Block size . . . . . 15
2.2.1 Hash algorithms . . . . . 15
2.2.2 Entropy . . . . . 16
2.2.3 Block sizes . . . . . 21
2.3 block-hashing . . . . . 21
2.3.1 Sliding block-hashing . . . . . 22
2.3.2 Aligned block-hashing . . . . . 23
2.3.3 Data Reduction in large datasets . . . . . 23
2.4 Existing block-hashing tools . . . . . 24
2.5 Verification . . . . . 26

**II Problem Statement and Methodologies 27**

**3 Problem Statement 28**
3.1 Is block hashing a recommended, sustainable method to identify presence of the reference data to use as admissible evidence in court . . . . . 28
3.1.1 Define criteria to ensure blocks in reference and target data are the same . . . . . 29
3.1.2 Setting bias for amount of mutual data between reference and target data . . . . . 29
3.1.3 Optimal block size to use . . . . . 29
3.1.4 Other factors to approve or disapprove the method as robust enough 29
3.1.5 Verifiable . . . . . 30
3.1.6 Is it feasible to combine the above criteria to ensure the technique produce admissible evidence . . . . . 30

**4 Research Methodology 31**

4.1 Creating datasets . . . . . 31
4.2 Collision probability . . . . . 32
4.3 Entropy bias . . . . . 33
4.4 Optimal block size . . . . . 33
4.5 Determine bias for coinciding and coherency of blocks . . . . . 34

**III Evaluation 35**

**5 Experimental setup 36**
5.1 Hardware . . . . . 36
5.2 Software . . . . . 37
5.3 Challenges in hardware/software and datasets . . . . . 37

**6 Experimental Sources 38**
6.1 Datasets . . . . . 38
6.1.1 The creation of the datasets . . . . . 39
6.1.2 Details about the datasets . . . . . 40

**7 Description of Results 45**
7.1 Results from dataset 1, msc database . . . . . 45
7.2 Results from dataset 2, msc_pictures database . . . . . 48
7.3 Results from dataset 3, msc_case database . . . . . 53
7.3.1 Detecting hash collisions in unallocated blocks . . . . . 54
7.3.2 Detecting equal blocks in reference data and unallocated areas . . 54
7.3.3 Detecting hash collision in blocks from unallocated areas . . . . . 56
7.3.4 Analysing hits between reference data and unallocated area . . . . 57
7.3.5 Connection between common blocks in unallocated areas and reference files . . . . . 63
7.4 Results from dataset 4, misc datasets . . . . . 77
7.4.1 Database msc_veracrypt . . . . . 77
7.4.2 Database msc_text . . . . . 77
7.4.3 Entropy in different file types . . . . . 80

**IV Discussion and Conclusion 82**

**8 Discussion 83**
8.1 Optimal hash algorithm to identify coinciding blocks . . . . . 84
8.2 Optimal Block Size to qualify the method . . . . . 84
8.3 Entropy to qualify the method . . . . . 86
8.4 Continuous blocks to qualify the method . . . . . 87
8.5 False positives/hash collisions . . . . . 93
8.6 Other factors with influence of the method . . . . . 94

8.7 The combination of block-size, entropy and continuous blocks . . . . . . . 96
8.8 Criteria for documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 98
8.9 Verification of findings . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
8.10 Other factors influencing the method . . . . . . . . . . . . . . . . . . . . . 100

**9 Conclusion 102**
9.1 Is block hashing a recommended, sustainable method to identify presence of the reference data to use as admissible evidence in court . . . . . . . . 103
9.1.1 Define criteria to ensure blocks in reference and target data are the same . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
9.1.2 Setting bias for amount of mutual data between reference and target data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
9.1.3 Optimal block size to use . . . . . . . . . . . . . . . . . . . . . . . . 105
9.1.4 Other factors to approve or disapprove the method as robust enough 106
9.1.5 Verifiable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
9.1.6 Is it feasible to combine the above criteria to ensure the findings are admissible evidence ? . . . . . . . . . . . . . . . . . . . . . . . . 108
9.2 Investigative skills . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
9.3 Further work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

**V Appendices 114**

**A Scripts 115**
A.1 Python Scripts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
A.1.1 Python script to create filehashes . . . . . . . . . . . . . . . . . . . . 116
A.1.2 Python script to create blockhashes . . . . . . . . . . . . . . . . . . 119
A.1.3 Python script to blockhash the case . . . . . . . . . . . . . . . . . . 125
A.1.4 Python script to remove duplicate files in video database . . . . . 127
A.1.5 Python script to calculate average color of the jpg pictures . . . . 129
A.1.6 Python script to convert JPG pictures to BMP-2 . . . . . . . . . . 131
A.1.7 Python script to blockhash the BMP-2 files . . . . . . . . . . . . . 133
A.1.8 Python script to create block maps of hits on references to unallocated . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
A.1.9 Python script to create block maps of blocks located in unallocated area of pre-existing files . . . . . . . . . . . . . . . . . . . . . . 136
A.1.10 Python script to calculate average entropy on file types . . . . . . 138
A.1.11 Python script to extract unallocated blocks from NTFS file system 138
A.2 SQL Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140

# List of Figures

2.1 Shannon entropy based on 2 bits (coin flip) . . . . . . . . . . . . . . . . . 17
2.2 Shannon entropy based on 8 bits . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.3 Shannon entropy based on 8 bits . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.4 Uniform distribution of probabilities . . . . . . . . . . . . . . . . . . . . . . 18
2.5 A perfect block with entropy of 1 . . . . . . . . . . . . . . . . . . . . . . . . 18
2.6 Calculation of uniform distribution of probabilities . . . . . . . . . . . . . . 19
2.7 Spectrum of Approximate Entropy Calculations [31] . . . . . . . . . . . . . 19
2.8 Principle sketch of sliding block-hashing . . . . . . . . . . . . . . . . . . . . 22
2.9 Principle sketch of aligned block-hashing . . . . . . . . . . . . . . . . . . . . 23
2.10 Block-hash search documentation in X-Ways Forensics . . . . . . . . . . . 25

7.1 Average color spread. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
7.2 Visualisation of the color spread. . . . . . . . . . . . . . . . . . . . . . . . . 49
7.3 Average entropy in JPG pictures per block-size. . . . . . . . . . . . . . . . 51
7.4 Average entropy in BMP-2 pictures per block-size. . . . . . . . . . . . . . 52
7.5 Block map of reference object 1, block size 4096 . . . . . . . . . . . . . . . 59
7.6 Block map of reference object 9, block size 4096 . . . . . . . . . . . . . . . 59
7.7 Block map of reference object 13, block size 4096 . . . . . . . . . . . . . . 60
7.8 Block map of reference object 20, block size 4096 . . . . . . . . . . . . . . 60
7.9 Block map of reference object 24, block size 4096 . . . . . . . . . . . . . . 60
7.10 Block map of reference object 1, block size 512 . . . . . . . . . . . . . . . 62
7.11 Block map of reference object 9, block size 512 . . . . . . . . . . . . . . . 63
7.12 Block map of reference object 13, block size 512 . . . . . . . . . . . . . . . 64
7.13 Block map of reference object 20, block size 512 . . . . . . . . . . . . . . . 65
7.14 Block map of reference object 24, block size 512 . . . . . . . . . . . . . . . 66
7.15 Block map of placement in unallocated areas object 1, block size 512 . . . 68
7.16 Block map of placement in unallocated areas object 9, block size 512 . . . 69
7.17 Block map of placement in unallocated areas object 13, block size 512 . . 70
7.18 Block map of placement in unallocated areas object 20, block size 512 . . 71
7.19 Block map of placement in unallocated areas object 24, block size 512 . . 72
7.20 Block map of placement in unallocated areas object 1, block size 4096 . . 73
7.21 Block map of placement in unallocated areas object 9, block size 4096 . . 74
7.22 Block map of placement in unallocated areas object 13, block size 4096 . . 74
7.23 Block map of placement in unallocated areas object 20, block size 4096 . . 75

7.24 Block map of placement in unallocated areas object 24, block size 4096 . . 76
7.25 Average entropy in AES encrypted container per block-size. . . . . . . . . 78
7.26 Average entropy in ASCII text per block-size. . . . . . . . . . . . . . . . . 79
7.27 SHA256 hash collisions and duplicate blocks in ASCII text per block-size. 80

8.1 Example of few spread blocks non sequential and non continuous . . . . . 88
8.2 Example of large chunks of common blocks found sequential and ordered in both the reference and target data . . . . . . . . . . . . . . . . . . . . . 89
8.3 block map of reference object 1, block size 4096 . . . . . . . . . . . . . . . 90
8.4 block map of placement in unallocated areas object 1, block size 4096 . . 91
8.5 block map of reference object 13, block size 4096 . . . . . . . . . . . . . . 91
8.6 block map of placement in unallocated areas object 13, block size 4096 . . 92
8.7 Data-reduction possible side-effects . . . . . . . . . . . . . . . . . . . . . . 95
8.8 Example of continuous blocks 512 vs 4,096 byte blocks . . . . . . . . . . . 97
8.9 Reconstruct with verification common block from reference and target data100

9.1 The block-hashing method illustrated. . . . . . . . . . . . . . . . . . . . . . 109

# List of Tables

4.1 Example records from dataset 2, the msc_picture filename-database . . . . 32
4.2 Example records from dataset 2, the msc_picture database using 512 byte blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

5.1 Setup of the Supermicro X9DAi computer . . . . . . . . . . . . . . . . . . 36
5.2 Setup of the Macbook Pro . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.3 Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

6.1 Overview of databases and their purpose . . . . . . . . . . . . . . . . . . 38
6.2 Number of different video types to block-hash . . . . . . . . . . . . . . . 40
6.3 Database: msc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.4 Database: msc_pictures . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
6.5 Database: msc_case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.6 Miscellaneous datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

7.1 Average entropy in vidoes . . . . . . . . . . . . . . . . . . . . . . . . . . 45
7.2 Detection of collisions in video file blocks in database msc . . . . . . . . . 46
7.3 Detection of collisions in JPG file blocks in database msc_pictures. . . . . 50
7.4 Detection of collisions in BMP file blocks in database msc_pictures. . . . . 50
7.5 Average entropy in jpg pictures on block-size. . . . . . . . . . . . . . . . 51
7.6 The table of reference files from KRIPOS. . . . . . . . . . . . . . . . . . 53
7.7 Collisions in blockhash_unallocatedNNN where NNN is the block-size. . . 54
7.8 Tables in cross query between reference blocks and from unallocated clusters. 55
7.9 Number of hits per reference file in the suspect,s unallocated area. Sorted by hits and block-size of 512 bytes. . . . . . . . . . . . . . . . . . . . . . 56
7.10 Number of hits per reference file in the suspects unallocated area. Sorted by hits and block-size of 4 096 bytes. . . . . . . . . . . . . . . . . . . . . 57
7.11 Average entropy in veracrypt AES encrypted container . . . . . . . . . . . 77
7.12 Average entropy in ASCII text . . . . . . . . . . . . . . . . . . . . . . . . 78
7.13 Collisions on different block sizes in msc_text database. . . . . . . . . . . 79
7.14 Average entropy on different file types. . . . . . . . . . . . . . . . . . . . 80

8.1 Detection of collisions in video file blocks in Dataset-1, block-size 4,096 . . 99

# Part I

# Introduction

# 1 Introduction

## 1.1 Motivation and Background

After computer forensics became a profession two decades ago, we now see that almost all types of criminal cases involve electronic evidence. Typically criminal cases with electronic evidence twenty years ago, were cases involving child abuse with sharing of pictures and videos of children. Other typical cases were fraud, counterfeit, abuse of data systems and a few other areas of criminal activity. Today, the typical case does not exist and all type of criminality involves electronic evidence. We now have a more broad range of data subject related to crime and there are alternatives to local stored pictures, videos and documents, but this kind of data objects still exists but often in a larger scale. Two decades ago, digital cameras/videos were not common. Today, this is something almost everyone have in the modern world.

Twenty years ago, a typical child abuse case involved a few hundred or maybe up to a few thousand media files, we now see cases with millions of pictures/videos [1]. In the Spanish pedophile operation "Penalty", 66 persons were charged for downloading 48 million child abuse pictures [2].

[1]http://legacy.king5.com/story/news/local/2014/08/02/13048690/

[2]http://www.eurekaencyclopedia.com/index.php/Category:Pedophilia_Law_Enforcement

A major problem investigating child abuse cases, is to connect the abuse material to the abuser (often the one taking the picture/video) and the victim. Many of the child abuse cases are initialized by world wide or national operations run by the central investigation units in one or more countries. Some of these cases targeting file sharing services on Internet (P2P networks). In Scandinavia there have been numerous such operations, like operation "Enea" where 153 Norwegians was charged for downloading through the P2P network FastTrack. In this operation the Police downloaded the abuse material the different offenders shared on this network. The police collected these files, made digital signatures of each file and distributed this to the local authorities where each offender is located.

As an investigator on some of these cases, one of the challenges was to connect the reference material to the offender. One reason could be that the suspect has erased the illegal material, another is that the person is innocent. If the evidence is erased a while ago, the chance of finding these pictures/videos by using carving techniques is very limited. While erased data content remains unchanged for a limited period of time, fragments of the data content remain for much longer, at least on storage where available free space is high. Carving techniques basically have the intention of recovering the whole object (file, document, video etc.).

This project involves identification of fragments from previous known data files with parameters to ensure we have fragments from the actual file and not another similar file.

The overall motivation for this project is to use identification parameters to compare pieces of data from a target source with pieces from a reference file by setting different criteria to ensure the pieces on the target is from the same object as the reference. Techniques to compare two equal sized blocks of data are well known and involve creating a digital signature of the two blocks and comparing these against each other. To create this signature we use well established hash algorithms like MD4, MD5, SHA-1, SHA-2, SHA-256, SHA-512 etc.

A challenge in this type of comparison is to exclude blocks we could expect to occur frequent in a dataset. Another challenge is to determine how many such equal blocks we need to have to ensure the fragment came from the same object as the reference material. Finally there are several sub parameters that could influence using this as a forensic method. One is the ability to make such comparison in a large scale related to a reasonable data-power available.

# 2 Litterature Survey

Several key elements are important in this research project. Factors such as forensic methodology, legal aspects, entropy, hash algorithms, block-size, block- hashing and verification. This chapter is divided into five key elements.

## 2.1 Forensic Methodology and Legal aspects

The general forensic methodology has nothing specificly about hashing blocks or piece wise search for data fragments. SANS Intitute has created a forensic methodology describing all steps in the investigation [6]. The block-hashing is not mentioned specifically but the importance of performing hash verification is throughout the paper. Most of the research on the block-hashing has a major focus on the technical aspects and is more described as a proof of concept. The relevant factors to have an approved methodology to perform such forensics is covered less. Different factors like block size, common blocks and hashing is more deeply covered. Entropy is just briefly covered.

None of the papers have a description or suggestion of how such a methodology should be when we come to quantity of evidence, quality of evidence and other factors to approve this as a forensic method. In an existing methodology, block-hashing could located in the part already covering carving and file signature analysis.

Block hashing is so far not found in any legal case referred to in my own country or elsewhere. According to general legal, the strength on an evidence rely on different factors. If we use some analogy here, we could compare this to a "normal" case where we have a suspect and a victim. At the crime scene, the police found different evidence. This evidence includes a shirt. A witness related to the suspect confirm that the suspect have exactly the same shirt as found on the crime scene. Same color, size, logo and the same vendor. This have no evidential meaning at all and is best treated as circumstantial evidence. In a city, several hundred could have the same shirt.

To become admissible evidence, the shirt from the crime scene must have other connections to the suspect. This is information such as DNA found on the shirt, spots located in certain places on the shirt, damage or other traces. This are the same questions we need examine in this project. We need to qualify and quantify the findings. It is not enough to confirm that two pieces of data are exactly equal from the reference material and the target object. On a normal data disk there are millions of blocks and many of these blocks are equal. A typical repeating block is a block of 512 bytes filled with equal values like only zero values (hexadecimal 0x00). This is a common block also in file objects. By comparing the reference blocks with the target blocks, we could end up having thousands of hits on typical repeating blocks.

This type of repeating block is also described as common blocks in [14] with reference to [12]. There is also such data blocks that is not that repeating, but we could find a few equal block common to both the reference and target data. This could be just a single block or just a few. To classify a hit between reference and target as admissible evidence we need to have some rules when it comes to quantity and quality.

In traditional forensics we perform hash verification on files and disks/disk images. When one file is hashed with a reliable hash algorithm compared to another file with same algorithm and have the same hash value, we are certain about the equality of these files, at least when it comes to files with a certain amount of content. Hash algorithms as a method to compare and ensure equality is not directly implemented in the the law books, but several articles have been written about the subject.

An article in the Indian "Magazine for Legal Professionals & Students", the use of hash algorithms is described [1]Authentication and Admissibility in Indian Perspective.

[1]http://lawyersupdate.co.in/LU/1/1288.asp

## 2.2 Hash algorithms, Entropy and Block size

### 2.2.1 Hash algorithms

The main identifier on the different blocks of data is a calculated hash value based on the data content.

> *Hashing is a powerful and pervasive technique used in nearly every examination of seized digital media. The concept behind hashing is quite elegant: take a large amount of data, such as a file or all the bits on a hard drive, and use a complex mathematical algorithm to generate a relatively compact numerical identifier (the hash value) unique to that data.* [29, p.9]

A hash is a digital signature of a data object and is an one-way calculation of the content. A hash value is distinct for an arbitrary object of data and can be used to compare if data objects are equal or not. Such a signature can only tell if two data objects are equal or not and can not tell anything about the content or how equal/unequal two or more objects are.

Two data-objects that are slightly different (just one bit or so) could have a very different hash signature. There exist numerous hash algorithms and one of the most known and most commonly used is the MD5 (Message Digest 5) [28] which is based on a 128-bit calculated hash value. Collisions in this were detected many years ago. A hash collision occur when two different set of data give the same hash value. The MD5 algorithm is not on the list of the US Department list of secure hashes [10].

Today, the more secure hashes are the SHA-1 [18], SHA-2 [32] and SHA-3 [20] algorithms. SHA-1 [18]is not longer on the same list of secure hashes and therefore not an option to use for this project. It is not always desired to use the most secure hash as the more bits it is based on, the more bytes each hash value has in each database record. Typically a SHA-256 will have a 64 byte string to represent the hash value while SHA-512 will occupy 128 bytes. SHA-2 is a family of four hash algorithms, the SHA-224/256/384 and SHA-512. The SHA-256 is the selected algorithm for this project. This is an algorithm stated as secure due to the FIPS 180-4 publication [10]. This algorithm has a theoretical probability of collision of $2^{256}$.

A collision occurs when two arbitrary data objects produce an identical hash. It is then possible to substitute one object for the other.

The dataset in one of the tables contains 15 billion records which is nearly $2^{44}$. This is a huge number of records and by using a strong algorithm, the probability of having a hash collision in two or more unequal objects should be reduced to a practical minimum. There is not any detected collisions so far on this algorithm. This is also the same algorithm used in the Bitcoin System [27]. The MD5 hash value is a 32 byte string, SHA-256 64 bytes and SHA-512 that is even more secure, each value is a 128 byte string. In a database with 15 billion records the MD5 will occupy $\approxeq$ 0.5 TB, SHA-256 $\approxeq$ 1 TB and SHA-512 $\approxeq$ 2 TB. This amount of differences in storage space is severe and need to been taken under consideration when deciding hash algorithms to implement. In order to compromise strength and storage requirements, SHA-256 was considered the best choice.

In the two papers and dissertation in [12, 14, 15] the MD5 hash is used to perform block-hashing. Other algorithms are briefly discussed but not implemented in the experimental work. In [15], page 29 there is some thoughts about the use of MD5 as a hashing algorithm for block-hashing as the algorithm is proved not to be collision resistant, but the reason for using MD5 is the wide use of MD5 in the forensic community. That was back in 2012 but that is still the fact in 2016. In [35] a method to break MD5 and other hashes. This work claims that breaking MD5 is done within 15 minutes and MD4 in few seconds. That was 11 years ago.

### 2.2.2 Entropy

Entropy is loosely defined as the randomness of a given data object and a measure of this randomness is given as the result of certain mathematical calculations. Entropy has been seen described as something to detect but entropy always exist on a data object. Entropy is not to be detected but to be measured. Entropy is not only a calculation used on data objects, but was initially used in thermodynamics and is followed by the first and second law of thermodynamics.

The entropy related to data information is based on the Shannon Entropy, [30] and from this, many entropy models are derived. Shannon entropy is much based on the Boltzmann entropy [5]. Shannon entropy is tailored to information theory while Boltzmann is more specific to thermodynamics. A more general explanation of different entropy models is described in [16].

There are four types of entropy:

1. Shannon (specific) entropy $H = -\sum_{i}^{N} log_2(\frac{i}{N})$

   H changes if you change how you express the data, i.e. if the same data is expressed as bit, bytes, etc H will be different. So you divide by log(n) where n is the number of unique symbols in the data (2 for binary, 256 for bytes) and H will range from 0 to 1 (this is normalized intensive Shannon entropy in units of entropy per symbol). But technically if only 100 of the 256 types of bytes occur n=100, unless you know the data generator had the ability to use the full 256. The value i is the number of times symbol occurred in N.

2. Normalized specific entropy: $\frac{H}{log(n)}$ Units are entropy (H)/symbol(log(n))

3. Absolute entropy $S = N \times H$

4. Normalized absolute entropy $S = \frac{N \times H}{log(n)}$

The entropy we will use on this project is the first one, the Shannon entropy with an entropy range from 0 to 1. A brief explanation in [33].

$$H = -\sum_{i=1}^{N=2} \rho_i log_2(\rho_i)$$

***Figure 2.1:*** *Shannon entropy based on 2 bits (coin flip)*

$$H = -\sum_{i=1}^{N=256} \rho_i log_{256}(\rho_i)$$

***Figure 2.2:*** *Shannon entropy based on 8 bits*

The two example formulas in 2.1 and 2.2 are the basic formulas. When we calculate entropy on data objects like files or blocks or data, the actual formula is the one in Figure 2.2. $\rho_i$ is the probability of the event i

The formula in Figure 2.1 is the general formula which calculates an entropy between 0 and 1 while the one in Figure 2.2 calculates entropy between 0 and 8. This is because the last one is based on 8-bit k values (each byte in a datao bject has 256 states, hexadecimal 0x00-0xFF).

The preferred range of presenting entropy is by using the original range from 0 to 1 and by converting the 8-bit to 1-bit, we will have the range between 0 - 1. This formula is shown in Figure 2.3

$$H = -\frac{1}{\sum_{i=1}^{N=256} \rho_i log_{256}(\rho_i)}$$

*Figure 2.3: Shannon entropy based on 8 bits*

We will show a few examples of entropy for some 512 byte blocks.

A block with uniform distribution of probabilities where $\rho_i = \frac{1}{N}$ that gives an entropy 1.0 is shown in Figure 2.7. This block has a perfect spread between all possible 8-bit values in the whole block and is explained with the following formula shown in Figure 2.4.

$$H = -\sum_{i=1}^{N}(\frac{1}{N})log_2(\frac{1}{N}) = log_2(N)$$

*Figure 2.4: Uniform distribution of probabilities*

```
0000000: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f  ................
0000010: 1011 1213 1415 1617 1819 1a1b 1c1d 1e1f  ................
0000020: 2021 2223 2425 2627 2829 2a2b 2c2d 2e2f   !"#$%&'()*+,-./
0000030: 3031 3233 3435 3637 3839 3a3b 3c3d 3e3f  0123456789:;<=>?
0000040: 4041 4243 4445 4647 4849 4a4b 4c4d 4e4f  @ABCDEFGHIJKLMNO
0000050: 5051 5253 5455 5657 5859 5a5b 5c5d 5e5f  PQRSTUVWXYZ[\]^_
0000060: 6061 6263 6465 6667 6869 6a6b 6c6d 6e6f  `abcdefghijklmno
0000070: 7071 7273 7475 7677 7879 7a7b 7c7d 7e7f  pqrstuvwxyz{|}~.
0000080: 8081 8283 8485 8687 8889 8a8b 8c8d 8e8f  ................
0000090: 9091 9293 9495 9697 9899 9a9b 9c9d 9e9f  ................
00000a0: a0a1 a2a3 a4a5 a6a7 a8a9 aaab acad aeaf  ................
00000b0: b0b1 b2b3 b4b5 b6b7 b8b9 babb bcbd bebf  ................
00000c0: c0c1 c2c3 c4c5 c6c7 c8c9 cacb cccd cecf  ................
00000d0: d0d1 d2d3 d4d5 d6d7 d8d9 dadb dcdd dedf  ................
00000e0: e0e1 e2e3 e4e5 e6e7 e8e9 eaeb eced eeef  ................
00000f0: f0f1 f2f3 f4f5 f6f7 f8f9 fafb fcfd feff  ................
0000100: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f  ................
0000110: 1011 1213 1415 1617 1819 1a1b 1c1d 1e1f  ................
0000120: 2021 2223 2425 2627 2829 2a2b 2c2d 2e2f   !"#$%&'()*+,-./
0000130: 3031 3233 3435 3637 3839 3a3b 3c3d 3e3f  0123456789:;<=>?
0000140: 4041 4243 4445 4647 4849 4a4b 4c4d 4e4f  @ABCDEFGHIJKLMNO
0000150: 5051 5253 5455 5657 5859 5a5b 5c5d 5e5f  PQRSTUVWXYZ[\]^_
0000160: 6061 6263 6465 6667 6869 6a6b 6c6d 6e6f  `abcdefghijklmno
0000170: 7071 7273 7475 7677 7879 7a7b 7c7d 7e7f  pqrstuvwxyz{|}~.
0000180: 8081 8283 8485 8687 8889 8a8b 8c8d 8e8f  ................
0000190: 9091 9293 9495 9697 9899 9a9b 9c9d 9e9f  ................
00001a0: a0a1 a2a3 a4a5 a6a7 a8a9 aaab acad aeaf  ................
00001b0: b0b1 b2b3 b4b5 b6b7 b8b9 babb bcbd bebf  ................
00001c0: c0c1 c2c3 c4c5 c6c7 c8c9 cacb cccd cecf  ................
00001d0: d0d1 d2d3 d4d5 d6d7 d8d9 dadb dcdd dedf  ................
00001e0: e0e1 e2e3 e4e5 e6e7 e8e9 eaeb eced eeef  ................
00001f0: f0f1 f2f3 f4f5 f6f7 f8f9 fafb fcfd feff  ................
```

*Figure 2.5: A perfect block with entropy of 1*

The block with entropy $= 1$ has two values (from hex 0x00 to 0xFF) through the whole block and we could try the formula on the block to show the way to the calculated normalized entropy. The calculation is shown in Figure 2.6.

$$H = -\frac{2}{512} \times log_{256}(\frac{2}{512}) - \frac{2}{512} \times log_{256}(\frac{2}{512}) - ........ - \frac{2}{512} \times log_{256}(\frac{2}{512})$$
$$H = 0.00391 + 0.00391 - ..... + 0.00391$$
$$H = 0.00391 \times 256 \approx 1$$

***Figure 2.6:*** *Calculation of uniform distribution of probabilities*

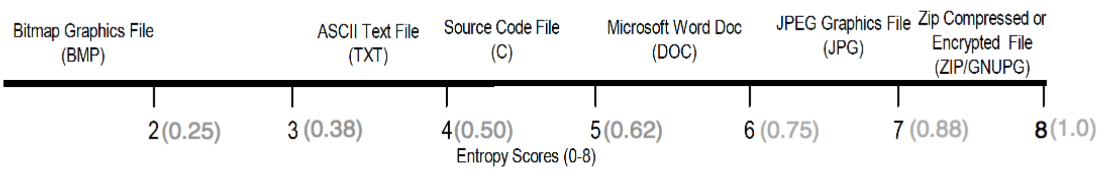


***Figure 2.7:*** *Spectrum of Approximate Entropy Calculations [31]*

As shown in Figure 2.7, entropy can tell us something about the data content which digital signatures cannot. Entropy is used in quantum mechanics related to energy and thermodynamics. The short explanation of entropy is found in Stanford Encyclopedia of Philosophy (http://plato.stanford.edu/entries/information/) and is described as: a measure of the amount of uncertainty.

In previous work, [12] entropy as a qualification on blocks is to some extent explained. that thesis uses the entropy value 0 to 8 is based on Shannon-entropy. In the same thesis, the use of high and low entropy is referred to, but no connection between entropy related to block size. In the same paper, the term high-entropy or low-entropy is not described. On a entropy scale from 0 to 1, it is unquestionable that 0 is low entropy and 1 is high-entropy. No literature was found to give a precise description on low, medium and high entropy.

In [14], some entropy tests are performed specifically related to common blocks but in a very low number of blocks. In [15] entropy is described in a general manner and there was also some test regarding common blocks but also a few examples of repeating blocks in PDF documents and in the OpenMalware 2012 dataset referred to from the article.

In [7] hash based carving was performed using graphic processors. The initial work found no false positives using MD5 and SHA-1 on a dataset of 528 million sectors. The same article focused on the smaller memory footprint lower strength hash algorithms produce.

### 2.2.3 Block sizes

In the prior work in [12, 14, 15] block sizes of 512 and/or 4,096 bytes have been used. Block sizes such as 1,024, 2,048, 8,192 and 16,384 are not mentioned.

Until recently the default sector size on hard drives has been 512 bytes. Still the size is widely used even if the 4,096 (4 KiB) byte sector-size is implemented as default size [9]. On Apple devices such as iPhone and iPad, sector sizes of both 8 KiB and 16 KiB are observed. Sector sizes are relevant mostly to whole disks but when we come to volumes on disk clusters or blocks are used. A cluster or block is one or more sectors concatenated together. Typically we find default cluster size of 4 KiB in the NTFS file system and the same default size in Apple HFS+ file system [8].

A phrase like "ideal block size" about a file system does not exist and usually is not required or desirable. Some volume set-up is fine tuned to a specific type of file to meet requirements related to speed and storage economy. If we look into the home marked segment, we might define 4 KiB as the ideal block-size since 95.5% of the operating systems in the desktop market is Windows or OS X based [2].

## 2.3 block-hashing

Traditionally, hashing of data objects is very common in the digital forensics environment, but mostly to create hashes of whole objects such as a full disk, disk image or file. block-hashing is a technique to create digital signatures from pieces of such objects [29]. In 2010 there was written an article "Garfinkel [13] that focused on using block hashes to perform sub-file forensics". Sub-file forensics is the same as block-hashing used in this project.

In [14] the term hash-based carving is used to describe the process of comparing equal size blocks from files with blocks from a data media. This term is a good description on the phase of comparing blocks, but the term is also a part of the whole block-hashing process. In 2015 the same technique is described as "Distinct Sector Hashes for Target File Detection" [14].

In this project, the terminology block-hashing is used. The use of "sub-file" in [13] and "Distinct Sector Hashes" [14] is technically the same but it‘s easy to relate these two terms to either files or disk sectors. block-hashing is not specifically related to a particular type of object and covers disks, files and other objects. In block-hashing the objects are divided into smaller pieces and earlier work on this subject uses either 512 or 4,096 byte.

[2]https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=0

There are other methods to perform block hashing such as fuzzy hashing which is a method to detect likeness specially related to malware [24]. In 2006 there was written an article "Kornblum [22] that focus on identifying almost identical files used context triggered piecewise hashing (CTPH)". The article also focused fuzzy hashing. This project has no intention of locating likeness but equality. The difference is demonstrated with the algorithms below where B is an arbitrary block of data. This project depend on $B_1 == B_2$ while fuzzy hashing depend on $B_1 \approx B_2$. A more in-depth discussion of fuzzy hashing is available in [19]. This project does not involve fuzzy hashing.

There are two major types of block-hashing, sliding block-hashing and aligned block-hashing.

### 2.3.1 Sliding block-hashing

In sliding block-hashing, each block-hash has the same size, but the blocks are not aligned to each other. This is demonstrated in Figure 2.8. Sliding block-hashes are described in [22] as "the rolling hash" and is basically the same thing.

In that example the block size is 64 byte and the sliding is 32 byte. This means that each block has 32 bytes from the previous block. The most intensive search using this method, is to set the sliding gap to one byte. By using the block size 64 as in the figure, first block is hashed from offset 0-63, next from 1-64, 2-65 ...$b_{S-64} - b_{S-1}$ where S is the size of the object.

Using this method could be very efficient, specially when we need to locate pieces of information not aligned in the file system to sectors or blocks. This method is particularly suitable for the location of malware that follows no such alignment to either sector, block or not even files.

The downside of this method is the amount of block-hashes which will increase by $\frac{Block-size}{Slidinggap}$ which in our example from Figure 2.8 provide a multiple of $\frac{64}{32} = 2$. A block size of 512 and sliding gap of 1 byte, gives us a multiply of $\frac{512}{1} = 512$.

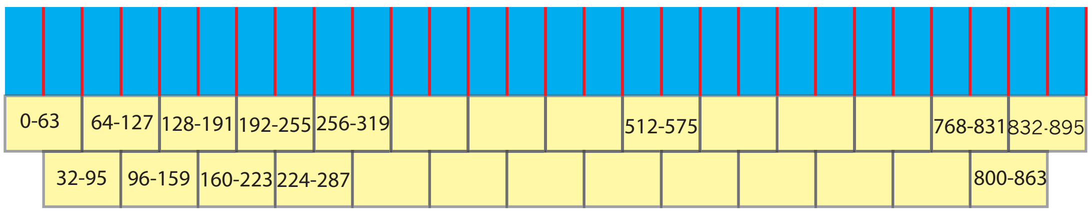


*Figure 2.8: Principle sketch of sliding block-hashing*

### 2.3.2 Aligned block-hashing

Block-hashing using alignment is the most obvious way of performing block-hashing on a file system where all data is minimum aligned to sector size (512 or 4096 bytes). The only data that is not aligned in a file system is the data inside objects like files. In that category, we also define files copied into containers as not aligned to the file system boundary.

Even if operating with aligning to 512 or 4096 byte sector size, there is no problem using multiples of these sizes and also fragments of these sizes. $\frac{2^n}{512}$

The principle of aligned block-hashing is shown in Figure 2.9 where we have a block size of 64 byte

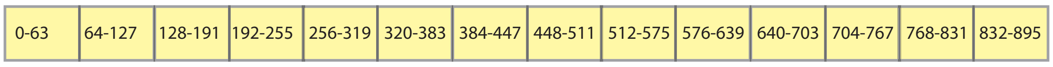


***Figure 2.9:*** *Principle sketch of aligned block-hashing*

The articles [14, 15] diverge between different block sizes and entropy is also mentioned, but not in depth. The article has no focus on determining bias for number of blocks/contiguous blocks in conjunction with entropy. In the thesis of Kristina Foster [12] there was done a huge work on similar challenges. The thesis was not focused on determining bias for entropy and the amount of hits to identify an object. In the two papers in [14] and [15] the block size is discussed and the conclusion is rather vague but it seems like a block size of 4,096 bytes is to prefer prior to 512 byte blocks.

### 2.3.3 Data Reduction in large datasets

Data reduction is a method to reduce the amount of data, usually in large datasets [25]. Typically, this is a method used in digital forensics to exclude known files from a set of files in a disk image. By performing hashing of files in a dataset and comparing the hash values with known files (files from known clean operating system installations, clean installations of software or using other criteria to reduce the number of files). Another method to perform data reduction, is to remove all files in a dataset with zero bytes (empty files).

This method could also be implemented in block-hashing forensics by reducing the number of hash values in the database. This could typically be hashes from blocks with certain known content like blocks filled with equal bytes (blocks with only 0x00, 0xff etc). Data reduction could also involve other criteria related to block-hashing. By eliminating all blocks with a very low entropy, the dataset could easily be reduced with a huge amount of records in the database of hash values. A combination of the above examples is also possible. In [25], different theoretical methods to perform data reduction are covered.

In [1] the method of using hash values to perform data reduction is covered and gives a brief introduction to that method with reference to the NIST collection of hash databases. The NIST NSRL (National Software Reference Library) is a large collection of hashes from known files. None of the data sets covers hash values from pieces of data (block hashes).

Another method to perform block-hash data reduction is to eliminate all hashes from allocated data in a large data set of files on a disk or disk image. In normal circumstances, we could assume that many files on a disk/disk-image have been copied, the original or copy have been erased and there is other forms of duplicates maybe erased at a certain time. Usually, when using the block-hashing technology to discover pieces of a reference file in a storage, we primarily search inside the unallocated area [2] described as an area on a disk not used by any file system object which could contain information from previous existing files or folders.

The allocated area is the opposite of the unallocated area and contain files, folders and other objects registered in the file system. This method of performing data reduction is not found in any reference material but is an extension of the commonly used ”known files” hash data reduction. Neither of the papers and master thesis in [12,13,15] discuss the data reduction topic in any context related to block-hashing.

## 2.4 Existing block-hashing tools

There are quite a very few sets of tools to perform block-hashing available.

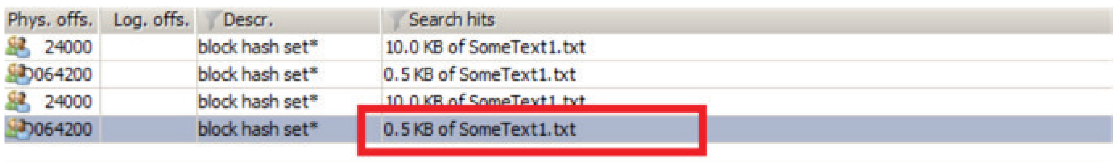
In the master thesis [12] the experimental work is primary based on the md5deep tool. This is a collection of tools that also support block wise hashing using different block sizes and algorithms. This tool is not optimized to support large databases [12].

Guidance Software is the vendor of the widely used forensic analysis tool, EnCase. The latest version is 7. As a bare application, there is limited functionality. By using the EnCase scripting language, Enscript, it is possible to create additional functionality for the tool. One of these scripts is the "File Block Map Analysis script" [21]. The technology is clear about hashing arbitrary number of reference files and compare it to hashes from a disk or volume. The script reports what fragments of the file is found. It is unclear how the script works, if there is any qualification of the different hits.

Another widely used forensic tool is the X-Ways Forensics too from X-Ways AG. This is a tool with in-built functionality, and what is described as block-hashing [11]. The technology is the same principle as described about Encase. X-Ways only support block hashes from a block size of 512 bytes.

Both Encase and X-Ways Forensics are closed source tools and the search technology is not available. When starting a block hash search in X-Ways, it claims that blocks with extremely high grade of repeating patterns are omitted. This could indicate use of entropy or similar technology to determine variation in the blocks. There is no documentation on what criteria are in use to exclude certain blocks.

The X-Ways Forensics is tested in a sharp case and the documentation and possibilities to ensure verifiability is covered to some extent. In Figure 2.10 we have an example of documentation. This example shown where some hits are located in the volume of the target data. There is no offset reference to where this hit relates to in the reference data.



| Phys. offs. | Log. offs. | Descr. | Search hits |
|---|---|---|---|
| 24000 | | block hash set* | 10.0 KB of SomeText1.txt |
| 064200 | | block hash set* | 0.5 KB of SomeText1.txt |
| 24000 | | block hash set* | 10.0 KB of SomeText1.txt |
| 064200 | | block hash set* | 0.5 KB of SomeText1.txt |

http://4.bp.blogspot.com/-UOqhUILKx-w/VWstk_qK2EI/AAAAAAAAA5A/jJilMs6nuLs/s1600/BlockHash4.png

***Figure 2.10:*** *Block-hash search documentation in X-Ways Forensics*

Brian Carrier refer to an article in the FBI forensic journal with the following text block.

> *Manufacturers of software used for image processing may be required to make the software source code available to litigants, subject to an appropriate protective order designed to protect the manufacturer's proprietary interests. Failure on the part of the manufacturer to provide this information to litigants could result in the exclusion of imaging evidence in court proceedings. This should be considered when selecting software.* [4, p.7]

The FBI article is published on the www.fbi.gov domain. [3]
Brian Carrier raises a principle question related to use of all forensic software and is very critical of closed source software. Today, the major investigating tools on computer forensic are closed source. Many of these tools have high credibility.

## 2.5 Verification

Verification in digital forensics is a well known term and implemented in several ways. Verification are often connected to hash-verification where an arbitrary amount of data is verified against a known hash to ensure validity. Research on hash verification and a proposed verification model is suggested in [23]. Verification is not implemented in all aspects of digital forensics and a secondary well established term is best-practise even if the the connotation of the word forensic infers a scientific method.

Different topics in digital-forensics have a lot of research, but mostly related to techniques and rarely on scientific tested methods.

Jason Beckett and Dr Jill Slay wrote an article in 2007, [3] which focus on validation and verification in a dynamic work environment. The article opens with the challenges in digital forensic work environment by high workload and less understanding to ensure quality with a high focus on "need for speed". The same article focus on the important principle of reproducibility, the ability to produce the same result using another tool or do the investigation by another person.

To implement this into this project, we need to make procedures in the method that ensure the principle of reproducibility. The research paper "Validation and verification of computer forensic software tools - Searching Function" [17] have the same focus on implementing paradigms that implement rules or procedures to ensure validity of findings.

[3]https://www.fbi.gov/about-us/lab/forensic-science-communications/fsc/july2001/swgit.htm

# Part II

# Problem Statement and Methodologies

# 3 Problem Statement

In this section we describe the research questions.

This project in focused on a single research question with several sub-questions needed to define a sustainable method to qualify block-hashing as a valid forensic method.

## 3.1 Is block hashing a recommended, sustainable method to identify presence of the reference data to use as admissible evidence in court

This is the main research question and it depend on the answers in the sub-questions. The goal of this project, is to describe a valid method to ensure evidence is sustainable and admissible in court of law. To ensure the findings are usable in court of law, we need to set several criteria to ensure parts of the reference data can be verified in the target data. The technique to make comparison between reference and target data is already tested in several research papers [12,14,15] by comparing MD5 hash values from one dataset with another using different block-sizes like 512 and 4,096 byte blocks.

### 3.1.1 Define criteria to ensure blocks in reference and target data are the same

This research question is basic in the block-hashing technique and is crucial to approve the quality of the findings. What kind of algorithms should be used to compare target blocks of data with blocks from the reference data. Prior works have used the MD5 hash algorithm entirely to compare equality. Is the use of a hash algorithm sufficient for comparison or should we combine the hash algorithm with additional algorithms like entropy ?

### 3.1.2 Setting bias for amount of mutual data between reference and target data

When we have decided what algorithms to use to compare blocks described in research question in 3.1.1, we need to have additional criteria to ensure the blocks in target data are from same data object as the reference data. This question is a measuring of quantity. To qualify the method, we need to have at least a certain amount of equal blocks of same size in the two datasets. The challenge is to define number of mutual blocks and how these blocks appears. Are the finding one or more chains of blocks or is the mutual blocks found spread as single blocks ?

### 3.1.3 Optimal block size to use

The evaluation of block-size raise many aspects in this method of localize partly erased evidence files. Is a small block-size (512 byte blocks) more reliable than using larger blocks like 4,096 bytes or visa versa ? Selecting a block-size, will this have an impact on the processing time ? Does the block-size have any unwanted side-effects ?

### 3.1.4 Other factors to approve or disapprove the method as robust enough

There are several other factors we need to evaluate. First of all, is it feasible to use this method to identify presence of blocks from reference data in a data target. Does processing power and time make the method useless in a practical case today with huge amount of data, even on the home segment. Are hardware and software critical elements in performing block-hashing as a method to detect mutual blocks ?

Is there any methods to reduce the amount of blocks, specially in the target data to make the block comparison feasible ?

### 3.1.5 Verifiable

By verifiable we mean the ability a third part have the ability to approve that the mutual data found in the target dataset are the same as in the reference data. This is one of the most important parts of an approved method to ensure the method is robust enough. What information do we need to make sure all finding are verifiable and to what extent?

### 3.1.6 Is it feasible to combine the above criteria to ensure the technique produce admissible evidence

By combining the criteria in Section 3.1.1 to Section 3.1.5, is it achievable to combine these criteria to ensure the finding can be used as admissible evidence in court ?

# 4 Research Methodology

The different research questions involves individual methods to answer and prior to answering any of the questions, datasets was made to perform different tests.

How the different datasets was created and the purpose of these is explained further in this chapter.

Each criteria (block size, entropy and coinciding blocks) are both separate parameters and have dependency on each other.

## 4.1 Creating datasets

The project involves four datasets that involves more than 8 TiB of data. The datasets are divided into equal pieces (blocks) of different sizes and the SHA256 hash of each block in addition to the block entropy is stored in databases. There is one database per dataset.

A typical database in the project is created using MySQL as the database engine and each database contains several tables with data records. One data record typically contains information about each block such as offset in the source data. The source is either a file-name or block number inside that source. In dataset 3, the msc_case database, is based on unallocated clusters. Here we only have reference to block number in the unallocated area.

Other data in the data set is the calculated SHA256 hash value that is a 64 byte string. In addition, we store the calculated entropy for that particular block. The SHA256 and the entropy value (1 bit entropy from 0 to 1).

| id | Filename | MD5 | Filesize |
|---|---|---|---|
| 12290 | G0010051.JPG | 39348e4ca5cd339c05b062e188c89d2b | 1382525 |
| 12291 | G0010052.JPG | 0dd76793066f449d1abb158a83a29f29 | 1387982 |
| 12292 | G0010053.JPG | 3aa5d9a8ea6434faef03e677f6e2f8b4 | 1388742 |

***Table 4.1:*** *Example records from dataset 2, the msc_picture filename-database*

| id | FileNum | SHA256 | BlockNum | Entropy |
|---|---|---|---|---|
| 1 | 12290 | f8bed49a070f836c49c95a002091 1f1360c9e075774fd6943a1ad120282a6b34 | 0 | 0.41879 |
| 2 | 12290 | 8fa4d109c03f39befa652385c143 fe98af9230f9184424bd23948578658ce84f | 1 | 0.17370 |
| 3 | 12290 | 003ab4d37db65e84e8e4cd261a59 f7256405f5dd090285b7559fabfc70fd190c | 2 | 0.34229 |

***Table 4.2:*** *Example records from dataset 2, the msc_picture database using 512 byte blocks*

The table 4.1 is used as lookup for the database shown in table 4.2 and is used as reference to the data-object ( could be other than files like disk, volume, unallocated area etc).

By using the database record set-up exemplified in Table 4.2 and 4.1 the need for verification is covered by using reference to file and block inside the file. The same method to process and document each block is used in all databases. In the database from an example case, there is no reference to any file as all blocks are from one source, the unallocated area and all block references are relative to the unallocated area.

## 4.2 Collision probability

A major part of this project is based on comparing block hashes inside a single database table or between several tables to identify equality or not. This method is based on processing hash values per block and comparing the reference block with the target block. To perform this, SQL queries are used between different tables and fields containing SHA256 hash values.

By using queries on a single database table, it is possible to detect block collisions in the same data set. By using queries between more than one table, it is possible to detect equality of data from two data sources.

By using databases with a very high number of records, the reliability of the experiments are higher than using smaller tables. With the phrase high number of records, we are speaking about several hundred million and not a few hundred thousand.

One of the databases in dataset 2, the msc_pictures is mainly used to generate data with a high amount of equality by taking more than 40,000 pictures of the same background. To demonstrate how equal the pictures are, an average color is generated. The intention with this database is not to create equal data but more to generate pictures that "look" equal to the human eye.

The largest data-set is dataset 1, the msc database with table records from almost 13,000 common videos of different format. The main purpose of this database is to create an extremely high number of records. One of the tables contains more than 15 billion records. Such large volumes of data are also an indicator of the challenges of big-data.

## 4.3 Entropy bias

With entropy bias we here speak about categorizing the amount of entropy. We will use the terms low, medium and high entropy. The challenge is to define these borders. We already know that entropy of 0 is low and 1 is high, and medium is not simply $> 0$ and $< 1$. To determine the bias of entropy, the testing was based on large amounts of database records from pictures, videos and a real case. By performing search for hash collisions in relation to the amount of entropy it is possible to observe the variation of entropy in large scale datasets.

By comparing entropy in different block sizes with same data as source is also a factor to decide the optimal amount of entropy used as a quality criteria in the block hashing methodology.

## 4.4 Optimal block size

To define an optimal block size, different block sizes were tested in databases with huge amounts of SHA256 hashes to determine collision frequency in conjunction with the time aspect of performing such searches. To determine an optimal block size, entropy is also part of the evaluation.

Several factors involve the optimal block size factor. One factor is the file-system environment another, the impact block-size has on entropy and finally the data processing efficiency.

Several block sizes are used but all sizes are a multiple of 512 bytes (512, 1,024, 2,048, 4,096, 8,192 and 16,384 bytes blocks)

## 4.5 Determine bias for coinciding and coherency of blocks

The block size and entropy parameters to use as qualification for the block hashing methodology are not enough to base the method on. Quantifying blocks is also an important factor to use to make the methodology robust enough.

As an example, just finding one single block of a certain size with a certain level of entropy is not enough to determine the presence of a previously complete data object in the target data.

The number of coinciding blocks is a crucial element in proofing the methodology.

In addition to the number of coinciding blocks, these blocks should have some coherence. An example of these two elements (coinciding blocks and coherence) is two blocks from a reference data object discovered in the target object. If the two blocks are found sequential, it could be a more certain identification than two blocks found spread in the target data.

To determine these factors, the dataset from a real case is used to show different grades of coincidence compared to coherence. This part of the test is done by using a SQL query between block hash records of target data and source data and determine coinciding blocks. This is the basic principle behind block hashing but also shows several examples using graphical block maps from the two data sets.

# Part III

# Evaluation

# 5 Experimental setup

This chapter describes the methods used to create the experiments.

## 5.1 Hardware

The processing of the datasets was performed using two machines, an Apple Macbook Pro 15" and a self build standalone machine with the following specifications.

| Component | Description |
|---|---|
| Mainboard | Supermicro X9DAi |
| CPU | 2 x Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz (6-core) |
| Raid controller | LSI MegaRAID SAS 9271-8i |
| Raid #1 | Raid-0, 6 TB (3 * 2 TB drives), SAS |
| Raid #2 | Raid-5, 8 TB (5 * 2 TB drives), SAS |
| Memory | Physical 80 GB RAM |
| System disk | OCZ 10xx SCSI Controller, PCI-X 8 card, 256 GB, SSD |
| Swap disk | CRUCIAL_CT256M225, 256 GB, SSD |
| OS | Windows 8.1 x64 |

***Table 5.1:*** *Setup of the Supermicro X9DAi computer*

| Component | Description |
|---|---|
| Machine basic | Apple Macbook Pro 15 inch, Retina, mid 2014 |
| CPU | 2.8 GHz Intel Core i7 |
| Memory | 16 GB |
| Disk | Apple SSD drive, 1 TB |
| OS | OS X 10.10 (Yosemite) |

***Table 5.2:*** *Setup of the Macbook Pro*

## 5.2 Software

One of the goals for this project, was using entirely free and open source tools to generate of the datasets and to perform the database search. Later in the project, the use use of MySQL in community version was not sufficient power to perform database queries on the largest datasets. In the final months of the project, we switch from MySQL to Microsoft SQL 2016.

| Type | Name and version |
|---|---|
| Database | MySQL 5.6.22 for Windows and Mac OS X |
| Database | Microsoft SQL 2016 Enterprise x64 |
| Database mgmt | MySQL Workbench |
| | Microsoft SQL Server Management Studio |
| Programming | Python 2.7.9 (Mac OS X) |
| | Python 2.7.11 (Windows) |

***Table 5.3:*** *Software*

## 5.3 Challenges in hardware/software and datasets

The three scenarios used in this project contains a huge amount of database data and the total number of record in the tree databases is nearly 17 billion records occupying nearly 2 TB of space.

The standalone Supermicro machine, Table 5.1 with 12 cores and 80 GB RAM is sufficient, but the MySQL 5.6.22 is the community version of the database and only support query on a single core and thread.

Some of the queries run several days, weeks and months to finish the processing using the MySQL database engine.

# 6 Experimental Sources

The research was based on large datasets created from four different test objects. The datasets are created with some fundamental values used to perform research against, the entropy and the hash algorithm, SHA256.

## 6.1 Datasets

| Dataset | Database | Information |
|---|---|---|
| 1 | msc | Database with block-hashes and other data from more than 12,000 videos of different format. |
| 2 | msc_pictures | Database with block-hashes and other data from more than 40,000 pictures of the same object. |
| 3 | msc_case | Database with hashes and other data from reference data the police claim the suspect have downloaded using P2P and block-hashes of the storage seized from the suspect. This is data from a real criminal case. |
| 4 | Misc | Miscellaneous datasets to test the probability of collisions in different type of data |

***Table 6.1:*** *Overview of databases and their purpose*

The purpose of dataset 1 is to provide large volume of hashed blocks to determine the probability of collisions when using different block sizes. This dataset is also used to determine an optimal amount of entropy to use when measuring the robustness of the forensic method.

The main purpose of dataset 2, is to provide a large amount of hashed blocks of different sizes from collection of data objects that visually identical. The block hashes are of different sizes and each picture remains in two states. The two states are the original lossy compressed JPG image and a simple uncompressed BMP-2 format. The last one is created by converting the JPG file to BMP using Python with the PIL extension. This dataset is also used to determine an optimal amount of entropy and optimal block-size to use when measuring the robustness of the forensic method.

Dataset 3 is from a real case where we have hashed and measured the entropy of blocks with different sizes. The data is two parts. One part is the reference data we would like to use to compare blocks from the suspects storage, the unallocated area. The intention of this dataset is to test the block-hashing on a real case and also to use it as a prototype to measure optimal block size and amount of entropy.

Dataset 4 is data from different file types. This dataset is mainly for testing entropy variation on different block-sizes.

By combining findings from the datasets, it should be feasible to determine the robustness of block-hashing as a forensic method by determining an optimal block-size, optimal amount of entropy and the amount of common blocks between a reference file and a search target.

### 6.1.1 The creation of the datasets

All database tables are created by importing tab or comma separated files with content generated from different Python scripts. A typical SQL script to import comma separated text into a table in a database is shown in listing A.12. The tab or comma separated text-file with the database records was created using Python scripts listed in A.1.

Initially, the Python scripts was set up to directly write to the MySQL database through the MySQL Python connector, but this was much slower than first write the data to CSV/TSV file and the use of the command similar to the one showed in A.12. The use of the SQL command "LOAD DATA INFILE" has an average import speed up to 250,000 records/second and the average speed parsing the dataset to CSV/TSV files from Python was around 50,000 records/second.

Initially, the MySQL database was set up to use the InnoDB database engine, but the huge amount of data records gives problems regarding the buffer pool of the transaction files and during the write process, the table fill gets slower and slower. When switched to MyISAM database engine (which is not transaction-based), the speed was higher and the problems with the buffer pool were solved. Later in the project, we switched to Microsoft SQL 2016 Enterprise database to have better performance against the largest data-sets. Some tests was not feasible to perform within a reasonable period of time. This is described more later in this part of the project.

### 6.1.2 Details about the datasets

In the following tables (6.3, 6.4 and 6.5), we have listed the statistics about the different databases in use. The three datasets are described briefly in the following sections. The three datasets contain of 20,781,267,668 records and occupies 2 TiB of storage when indexes are included.

#### 6.1.2.1 Dataset 1, Database:msc

This is the largest test set containing data from 12,289 videos. Initially, this dataset contains some more videos, but with some duplicates detected by the MD5 checksum. The Python script in A.1.4 removes duplicates from the file name table.

The number of different video files involved in this dataset is specified in Table 6.2

| File type | Number of files |
|---|---:|
| avi | 7,506 |
| mkv | 3,582 |
| mpg | 418 |
| mp4 | 396 |
| wmv | 45 |
| vob | 246 |
| Other types | 96 |
| Sum | 12,289 |

***Table 6.2:*** *Number of different video types to block-hash*

The Python script in A.1.2 was used to generate the block hashes from the videos. There was generated block-hashes of 512 and 4,096 bytes blocks.

Some data-objects in this dataset has a file size not aligned to 512 and/or 4,096 bytes. These files are referred to in the filenames512 and filenames4096 respectively. The last block on file which has no alignment to the block sizes set has a values of 1 in the "remnant" field.

Check the Table 6.3 with details about the number of records in each table.

| Table | Engine | Avg row length in bytes | Rows | Created time |
|---|---|---:|---:|---:|
| blockhash4096 | MyISAM | 95 | 1,950,241,553 | 15.01.16 00:17 |
| blockhash512 | MyISAM | 95 | 15,601,968,046 | 20.02.15 09:09 |
| filenames4096 | MyISAM | 7 | 12,289 | 16.01.16 21:10 |
| filenames512 | MyISAM | 7 | 12,289 | 16.01.16 21:11 |
| hashdatabase | MyISAM | 145 | 12,289 | 20.02.15 09:09 |
| SUM rows | | | 17 552 246 466 | |

***Table 6.3:*** *Database: msc*

#### 6.1.2.2 Dataset 2, Database:msc_pictures

This dataset is based on pictures taken with a GoPro Hero 3+ Black edition camera in time-lapse mode with 2 pictures/second. The object for the pictures was some standard white paper sheets pinned to a wall in a room with normal lightning. The camera was fitted with a Sandisk 64 GB memory card and the GoPro was put on a stand to take steady pictures automatically of the white sheets until the memorycard was full and the result was 40,493 pictures. The size of each picture was from 1,373,383 to 1,406,842 bytes and the pictures has the pixelsize (w/h) of 2,560 * 1,920. The pictures were taken between 12. Feb 2015 12:00:04 and 19:37:17 which is 7 h 37 m 13s. The picture format was JPG.

A database, msc_pictures was created with file-names and different block-hashes in the range of 512, 1024, 2048, 4096 and 8192 bytes. Number of records in the different tables in the msc_pictures is shown in Table 6.4.

| Table | Engine | Avg row length in bytes | Rows | Created time |
|---|---|---:|---:|---:|
| hashdatabase | MyISAM | 136 | 40,493 | 20.01.16 12:17 |
| blockhash512 | MyISAM | 95 | 109,749 164 | 17.02.15 14:35 |
| filenames512 | MyISAM | 7 | 40,493 | 19.01.16 14:44 |
| blockhash1024 | MyISAM | 95 | 54,864 493 | 20.01.16 12:00 |
| filenames1024 | MyISAM | 7 | 40,493 | 20.01.16 12:00 |
| blockhash2048 | MyISAM | 95 | 27,422 138 | 20.01.16 12:00 |
| filenames2048 | MyISAM | 7 | 40,493 | 20.01.16 12:00 |
| blockhash4096 | MyISAM | 95 | 13,700 750 | 17.02.15 14:35 |
| filenames4096 | MyISAM | 7 | 40,493 | 19.01.16 14:44 |
| blockhash8192 | MyISAM | 95 | 6,839 546 | 20.01.16 12:00 |
| filenames8192 | MyISAM | 7 | 40,493 | 20.01.16 12:00 |
| blockhash512_raw | MyISAM | 103 | 1,166,198,400 | 13.02.16 11:33 |
| blockhash1024_raw | MyISAM | 103 | 583,099,200 | 13.02.16 11:54 |
| blockhash2048_raw | MyISAM | 103 | 291,549,600 | 13.02.16 11:54 |
| blockhash4096_raw | MyISAM | 103 | 145,774,800 | 13.02.16 11:53 |
| blockhash8192_raw | InnoDB | 128 | 65,518,105 | 09.02.16 15:16 |
| Sum rows | | | 2,464,959,154 | |

***Table 6.4:*** *Database: msc_pictures*

#### 6.1.2.3 Dataset 3, Database:msc_case

This dataset is based on a real case with a seized disk from the suspect‘s machine and some videos the National Investigation Authority of Norway claimed the suspect downloaded at a certain point of time. To extract all unallocated blocks from the NTFS volume on the suspects disk the Python script A.1.11 is used. The result is a raw data image file. The same data is generated with the Sleuthkit *blkls* application. Both give the same result. The raw image of the unallocated clusters is the source to generate table records using different block sizes ( 512, 1,024, 2,048 and 4,096 bytes blocks). This dataset also contains block-hash data from the reference data that consists of 26 different videos. block-hash data from the 26 videos are also generated with the same block size as the target data.

| Table | Engine | Avg row length in bytes | Rows | Created time |
|---|---|---:|---:|---:|
| blockhash4096 | MyISAM | 91 | 36,330,752 | 20.02.15 09:03 |
| blockhash512 | MyISAM | 91 | 290,646,016 | 20.02.15 09:03 |
| blockhash_unallocated1024 | MyISAM | 91 | 117,273,276 | 29.01.16 20:56 |
| blockhash_unallocated2048 | MyISAM | 91 | 58,636,638 | 29.01.16 20:56 |
| blockhash_unallocated4096 | MyISAM | 91 | 29,318,319 | 06.02.16 16:11 |
| blockhash_unallocated512 | InnoDB | 111 | 216,596,681 | 07.02.16 16:24 |
| reference_blockhash1024 | MyISAM | 95 | 3,623,051 | 11.02.16 18:32 |
| reference_blockhash2048 | MyISAM | 95 | 1,811,521 | 11.02.16 18:32 |
| reference_blockhash4096 | MyISAM | 95 | 905,755 | 06.02.16 16:18 |
| reference_blockhash512 | MyISAM | 95 | 7,246,112 | 06.02.16 19:42 |
| reference_hashdatabase | MyISAM | 150 | 26 | 11.02.16 18:23 |
| caselookup4096 | MyISAM | 91 | 18,877 | 09.02.16 18:34 |
| caselookup512 | MyISAM | 91 | 163,323 | 08.02.16 17:48 |
| reference_hashdatabase | MyISAM | 152 | 26 | 11.02.16 15:11 |
| unallocated_collisions4096 | MyISAM | 83 | 1,491,675 | 12.02.16 17:05 |
| unallocated_collisions512 | MyISAM | 83 | NIL | NIL |
| SUM rows | | | 764,062,048 | |

***Table 6.5:*** *Database: msc_case*

#### 6.1.2.4 Dataset 4, miscellaneous data sets

| Database | Table | Engine | Avg row length in bytes | Rows | Created time |
|---|---|---|---|---|---|
| msc_text | blockhash_all | MyISAM | 95 | 1,040,731 | 23.02.16 12:20:59 |
| msc_veracrypt | blockhash_all | MyISAM | 95 | 103,219,200 | 23.02.16 10:55:52 |

***Table 6.6:*** *Miscellaneous datasets*

# 7 Description of Results

This chapter describes the outcomes of applying the approach to solve the research problems formulated in Chapter 3.

Here we will describe the results of the different experiments and relate them to proposed research questions.

## 7.1 Results from dataset 1, msc database

The main purpose of this dataset is to test the probability of hash collisions using large number of records. This data-set is also used to determine average entropy related to different block sizes.

The average entropy for the two tables is shown in Table 7.1. The entropy has the same curvature as tested in the other datasets (msc_pictures and msc_case) but only tested on block-size 512 and 4,096 bytes.

| Block size | Entropy |
|---:|---|
| 512 | 0.939041 |
| 4096 | 0.986328 |

***Table 7.1:*** *Average entropy in vidoes*

The average entropy in dataset 1 is shown in Table 7.1. The entropy on 4,096 byte blocks are significantly higher than in 512 byte blocks. Since we only have tested two block-sizes on this dataset, the graph is omitted and would not have shown the correct picture on how the entropy increases as the block-size increase. The main goal of this dataset was to detect hash collision and not entropy.

Collision detection was initially done using MySQL on the 4 KiB block-size. The processing was aborted after 87 days and the same dataset was created in a Microsoft SQL Enterprise 2016 database engine. The collision results are shown in Table 7.2.

| **Block-size** | Block Collisions | Block Collisions % | Hash Collisions | Hash Collision % | Proc time min | Entropy Avg | Max | Min |
|---|---|---|---|---|---|---|---|---|
| 512 | 59,473,671 | 0.375 | 7,220,501 | $4.6 \times 10^{-5}$ | 660 | 0.907 | 0.968 | 0.000 |
| 4,096 | 4,899,001 | 0.250 | 623,504 | $3.2 \times 10^{-5}$ | 44 | 0.936 | 0.996 | 0.000 |

***Table 7.2:*** *Detection of collisions in video file blocks in database msc*

Table 7.2 shows the statistics on the hash collisions in the msc dataset from 12,289 videos of different type. We distinguish between block and hash collisions. The block-collisions are total number of blocks involved in collisions which has a minimum of two blocks per hash collision. The hash-collisions are the number of SHA256 hashes we found collisions on. From the numbers in this table we notice that the following relation. $\frac{block-collisions}{hash-collisions} \approx 8$. This means the average hash collision involves an average of about 8 blocks.

From the Table 7.2, we notice that difference in number of collision blocks in 4,096 byte blocks are severely lower than the 512 byte blocks. The difference is ≈ 50%.

In the largest database table has 15,549,714,874 unique SHA256 hash values out of 15,601,968,046.

The difference in number of hash collisions are severe. The collision probability in this data-set between $\frac{Blockssize4096}{Blocksize512}$ is $\frac{1}{700}$.

Collisions in video files, specially MP4, AVI and other are normal due to the architecture of these files. As an example, AVI files [26] uses areas in the file for indexes, sub-indexes and super-indexes. In addition, it's not uncommon to find areas with "garbage" to set a delay in the video.

In the msc dataset from video files with 4,096 byte blocks, there are several examples of such content in the video. In one file with a size of 740 MB, large areas of different indexes was found. One particular block was repeated 240 times and the content appears to be indexes of some kind.

In the Matroska (MKV file name suffix) we also find number of repeating blocks.

Common to these repeating pattern-filled-blocks in this type of files are the amount of entropy. Several tests on these repeating patterns which give an entropy between 0.2 to 0.4 with some exceptions in some of the AVI video files.

Entropy generally are very high on video files. The average entropy for the whole msc data set of 4,096 byte block size is 0.9362. If we exclude all entropy below or equal to 0.5000 we end up having an average of 0.9888.

Further searches in the database show only 9,217 colliding blocks with entropy $> 0.9$ with more than 2 hits. About 1 million collisions in blocks with entropy $> 0.9$ with only two hits.

Many of the videos in the collection about 13,000 files was from TV-series and other series. Common to many of the TV-series are the use of introduction vignettes with both sound and picture. If these objects are block aligned in the file, it will be normal to have several hits on these vignettes.

After working with child abuse cases many years, it is not uncommon to have such vignettes in this type of data to "credit" the author.

As shown in Table 7.2, we had a total of 4.8 million collisions. 1.3 million of these are blocks with an entropy of 0.0. With a database of nearly 2 billion records, this is only a total collision of 0.24%. If we exclude the blocks with zero entropy, we ends up with a collision of 0.125%.

The column "proc time" is the SQL processing time to perform the comparison using Microsoft SQL.

## 7.2 Results from dataset 2, msc_pictures database

The main purpose of this dataset is to test the probability of collisions in data made to look equal to the human eye. This is more than 40,000 pictures taken with a steady mounted camera taking time lapse pictures. The photo background is static and the pictures are taken inside a closed room with minimal impact from external activity. The purpose was not to make the pictures technical absolute equal, but equal to the human eye. The picture data format is JPG which is a lossy-compressed picture format. The same pictures was also converted to BMP-2 picture format to perform similar searches on. The first calculations on this dataset is to proof the equality of the pictures. This is measured by the comparing colours of the dataset to show the relative low variation in the visual content.

The average entropy for 512 byte block-hashes was 0.9413 and 0.9416 for blocks with entropy $> 0$. There were 40493 records with entropy 0. That were one per file, and all was on block 4 (offset 1,536-2,047). This block on all pictures contains a repeating pattern of hexadecimal 0x00 and was part of the meta-data in the picture. Meta-data in JPG pictures are data that defines the picture set-up with picture size, color depth, white balance, number of colors, EXIF[1] data and much more. On a typical picture taken with a standard digital camera, the meta-data often occupies the first two-four KB.

The average color per picture and average color between all pictures was also calculated. The processing of average color was done using Python and the PIL package. The script is in appendix A.1 and is the script A.1.5.

The pictures was in RGB (Red, Green, Blue) color mode and the Red was between 132-137, Green between 120-124 and Blue between 114-120.

The average color in RGB is (134,122,117) is decimal the color code $8781824_{10}$, hexadecimal $867A75_{16}$.

The average color is visualized in the graph, shown in Figure 7.1

Figure 7.1 shows the decimal color number (Y-axis) on each picture (X-axis in the picture number). All these calculations on the pictures, were made to show how little each picture differs from each other. The only impact on the pictures, was the amount of light that came through a small window just below the roof. Otherwise, there was no activity in the room and there was no traffic too. There is a small drop near picture 15,000 and has probably to do with less light from the small window near the roof. The small variation from low, average to highest color number is almost not visible to the human eye. The difference is illustrated in Figure 7.2. There is tree coloured rectangles in the figure and each represent the the colour variations (high, average and minimum) in this dataset.

[1]http://www.media.mit.edu/pia/Research/deepview/exif.html

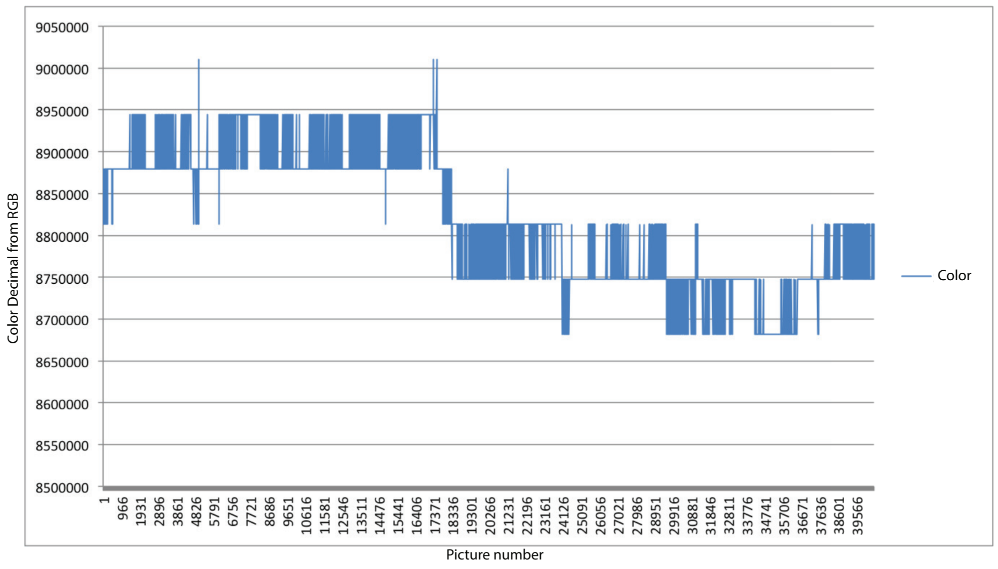


***Figure 7.1:*** *Average color spread.*

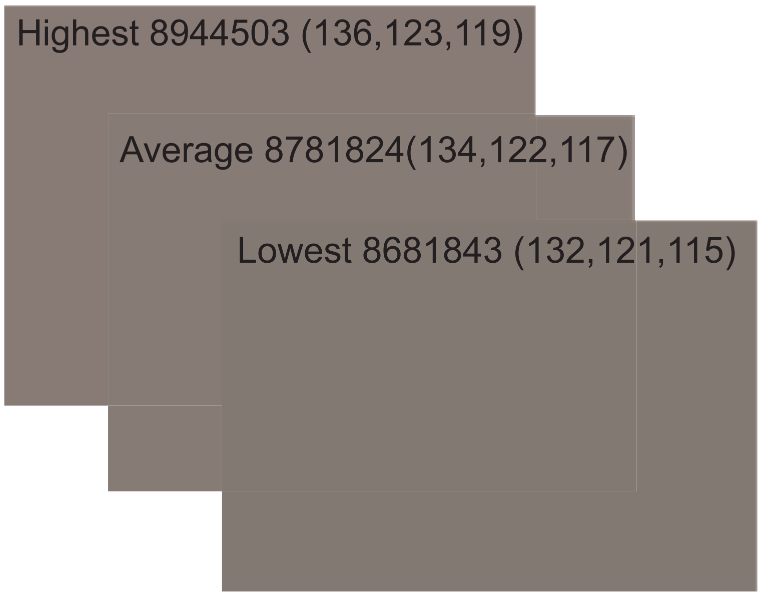

Highest 8944503 (136,123,119)
Average 8781824(134,122,117)
Lowest 8681843 (132,121,115)

***Figure 7.2:*** *Visualisation of the color spread.*

After demonstrating the equality of the pictures in the dataset, we perform a SQL query in the dataset to detect block collisions in the dataset. The data we search inside, is the tables with block-hash data (SHA256 hashes and entropy) per block-size (512, 1,024, 2,048, 4,096 and 8,192 byes). We both search in the blocks from the JPG format of the files and the BMP-2 format of the same after been converted. The SQL query in code A.13 was used to detect duplicate SHA256 in a table.

Collisions in the database of .jpg picture hashes, msc_pictures, Table 7.3 and 7.4.

| Table block-size | Blocks w/ Collisions | Search time in minutes | Entropy Avg | Entropy Max | Entropy Min |
|---|---|---|---|---|---|
| blockhash512 | 162,292 | 260 | 0.427590144 | 0.826447493 | 0.0000 |
| blockhash1024 | 40,405 | 120 | 0.312801256 | 0.313755647 | 0.198011959 |
| blockhash2048 | 0 | 49 | · | · | · |
| blockhash4096 | 0 | 9 | · | · | · |
| blockhash8192 | 0 | 2 | · | · | · |

***Table 7.3:*** *Detection of collisions in JPG file blocks in database msc_pictures.*

Table 7.3 shows the collisions in the JPG pictures in the dataset. The left most column is the table name with block-size. Table *blockhash512* is the blocks of 512 bytes. The next column is the number of blocks with collisions. The three last columns is the average, maximum and minimum entropy of the blocks with collisions. On the three last rows, there was not detected any collisions (block-size 2,048, 4,096 and 8,192). Entropy given with a "dot" indicates no data.

| Table block-size | Collisions | Search time in minutes | Entropy Avg | Entropy Max | Entropy Min |
|---|---|---|---|---|---|
| blockhash512_raw | 0 | 13,212 | · | · | · |
| blockhash1024_raw | 0 | 6,393 | · | · | · |
| blockhash2048_raw | 0 | 3,264 | · | · | · |
| blockhash4096_raw | 0 | 524 | · | · | · |
| blockhash8192_raw | 0 | 53 | · | · | · |

***Table 7.4:*** *Detection of collisions in BMP file blocks in database msc_pictures.*

Table 7.4 shows collisions on blocks from the same pictures converted to BMP-2. The query against the block-hashes give no collisions in any of the block-sizes.

We have also calculated the average entropy on both JPG and BMP-2 file blocks of different sizes. These values are shown both in a table and as a graphic plot. Table 7.5 covers both the JPG and BMP-2 pictures. Fig 7.3 are from the calculations on JPG pictures and fig. 7.4

| block-size | Avg entropy JPG | Avg entropy BMP-2 |
|---:|---:|---:|
| 512 | 0.9413 | 0.5244 |
| 1024 | 0.9684 | 0.5520 |
| 2048 | 0.9810 | 0.5893 |
| 4096 | 0.9871 | 0.6318 |
| 8192 | 0.9901 | 0.6623 |
| AVG | 0.9736 | 0.5920 |

***Table 7.5:*** *Average entropy in jpg pictures on block-size.*

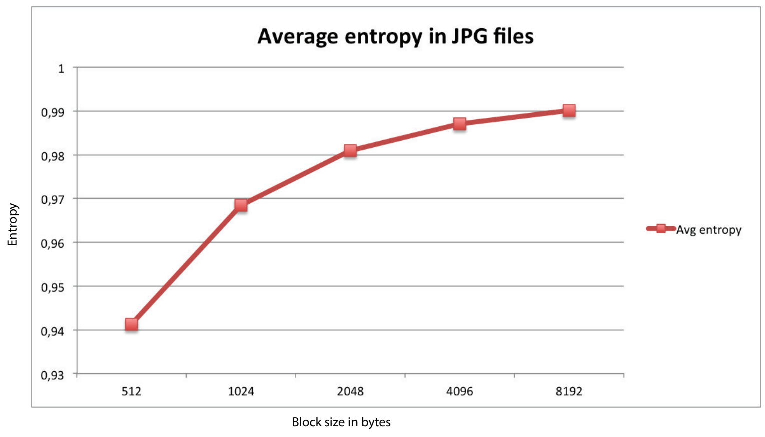


***Figure 7.3:*** *Average entropy in JPG pictures per block-size.*

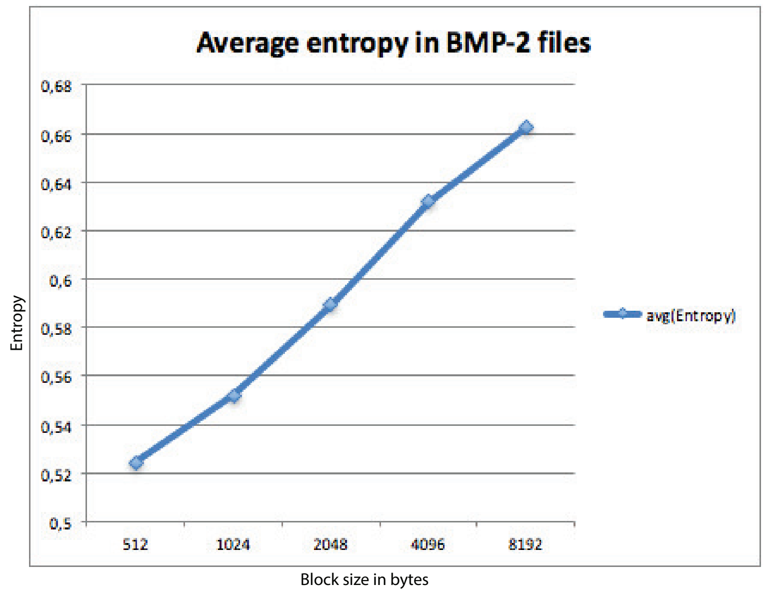


*Figure 7.4: Average entropy in BMP-2 pictures per block-size.*

## 7.3 Results from dataset 3, msc_case database

Table 7.6 provides information about reference data delivered by KRIPOS (National Criminal Investigation Service, Norway) from an active case. This contains 26 objects showing sexual abuse of children. KRIPOS claim the suspect downloaded these videos from a P2P network at a particular time. None of these files were found as active files on the suspect's storage. The files used as reference data i real child abuse videos and the file-names in the table are anonymized.

| File number | Filename Name | File size |
|---:|:---|---:|
| 1 | [–anonymized–].avi | 537,726,976 |
| 2 | [–anonymized–].avi | 2 904,678 |
| 3 | [–anonymized–].mpeg | 25,159,442 |
| 4 | [–anonymized–].mpeg | 401,113,088 |
| 5 | [–anonymized–].avi | 329,743,512 |
| 6 | [–anonymized–].mpeg | 20,301,246 |
| 7 | [–anonymized–].mpeg | 34,336,776 |
| 8 | [–anonymized–].avi | 77,248,512 |
| 9 | [–anonymized–].avi | 230,114,636 |
| 10 | [–anonymized–].avi | 60,688,896 |
| 11 | [–anonymized–].mpeg | 134,029,577 |
| 12 | [–anonymized–].avi | 99,159,302 |
| 13 | [–anonymized–].avi | 75,805,306 |
| 14 | [–anonymized–].avi | 356,837,650 |
| 15 | [–anonymized–].mpeg | 38,496,260 |
| 16 | [–anonymized–].avi | 47,024,784 |
| 17 | [–anonymized–].avi | 47,024,784 |
| 18 | [–anonymized–].avi | 157,095,936 |
| 19 | [–anonymized–].mpeg | 16,384,000 |
| 20 | [–anonymized–].mpeg | 52,162,560 |
| 21 | [–anonymized–].mpeg | 438,836,873 |
| 22 | [–anonymized–].mpeg | 45,620,923 |
| 23 | [–anonymized–].avi | 35,205,120 |
| 24 | [–anonymized–].mpeg | 151,721,296 |
| 25 | [–anonymized–].avi | 117,831,384 |
| 26 | [–anonymized–].mpeg | 177,439,724 |
|  | SUM | 3,710,013,241 |

***Table 7.6:*** *The table of reference files from KRIPOS.*

### 7.3.1 Detecting hash collisions in unallocated blocks

| Database table | SHA256 collisions | Non zero blocks Involved | Search time in min | Note |
|---|---|---|---|---|
| blockhash_unallocated512 | 11,791,229 | 50,544,798 | 817 | 8,953,059 of the blocks involved is from the block with SHA256: 076a27c79e5ace2a3d47f9 dd2e83e4ff6ea8872b3c2218f66c92b89b55f36560 and entropy of 0.0. This is blocks with only 0x00 values |
| blockhash_unallocated1024 | . | . | . | . |
| blockhash_unallocated2048 | 2,908,065 | 12,069,348 | 106 | 1,963,916 of the blocks involved is from the block with SHA256: e5a00aa9991ac8a5ee3109844 d84a55583bd20572ad3ffcd42792f3c36b183ad and entropy of 0.0. This is blocks with only 0x00 values |
| blockhash_unallocated4096 | 1,491,675 | 5,696,144 | 53 | 823,877 of the blocks involved is from the blocks with SHA256: ad7facb2586fc6e966c004d7 d1d16b024f5805ff7cb47c7a85dabd8b48892ca7 and the entropy is 0.0. This is blocks with only 0x00 values |

***Table 7.7:*** *Collisions in blockhash_unallocatedNNN where NNN is the block-size.*

In Table 7.7 the column "SHA256 collisions" is the number of SHA256 collisions in the table. The column "Non zero blocks involved" is the sum of SHA256 collisions per row.

Detecting a large number of collisions in unallocated areas is common, in particular with disks not run for a very long time. Such disks will have large areas of blocks with only zero-values, 0x00.

### 7.3.2 Detecting equal blocks in reference data and unallocated areas

In this section we will perform a lookup of block-hashes from reference files into the block-hashes from unallocated areas. This is a practical example of the block-hashing technique in general. The search is based on blocks with common SHA256 values. The result of the query will be a table of records from the two data areas, and contains the common data in addition to where the common blocks are found in both data areas. As an example we could have a row with information about where in the reference material the hash remain in, where in the unallocated areas the same hash/hashes remain in and the common data like the SHA2556 and the amount of entropy. One hit from the reference data could have several hits from the unallocated areas.

The different tables in the msc_case database are shown in Table 6.5. In Table A.14 we have number of hits from the reference table in the table of block-hashes of the unallocated area of the suspects storage. One table per blocksize ( 512 and 4 096 byte blocks). This is shown in the two tables, Table 7.9 and 7.10. The File number refers to Table 7.6.

A CSV file of mutual SHA256 hashes from the reference material and the unallocated area of the suspects storage was generated using the SQL query shown in code A.14. This CSV file was used to create new tables containing only hits between reference data and the unallocated clusters on the suspects storage. This new table are based on query between the two databases (reference data and unallocated areas). We then have a single table with relatively small number of records to perform further searches in.

The cross search performed using the Python code in A.14 was between the two tables which give us the result in the third table shown in Table 7.8. The table shows the number of common blocks in the new table with common hits. With the newly created table, we just have a limited number of records to perform further searches in. The SQL query example are for 512 byte block tables.

| | Table in msc_case | Records |
|---|---|---:|
| 1 | reference_blockhash512 | 7,246,112 |
| 2 | blockhash_unallocated512 | 234,546,552 |
| 3 | caselookup512 | 163,323 |
| 4 | reference_blockhash4096 | 905,755 |
| 5 | blockhash_unallocated4096 | 29,318,319 |
| 6 | caselookup4096 | 18,877 |

***Table 7.8:*** *Tables in cross query between reference blocks and from unallocated clusters.*

Table 7.8 have to major row sections. Row 1 to 3 for cross-searches between 512 byte blocks. First row is reference data, next is same sized blocks from unallocated areas and the third is the table with common blocks from the two previous. To evaluate number of hits per file, the only table involved is the caselookup[blocksize] (like caselookup512)

By using the SQL query in appendix A.15 we get the following numbers of hits between the reference data and the unallocated clusters per reference file, shown in Table 7.9 and 7.10. The "File numbers" is the one listed in 7.6.

| File number | File Size | Hits | Percent hits |
|---|---|---|---|
| 13 | 75 805 306 | 57,508 | 38.84 % |
| 9 | 230 114 636 | 43,274 | 9.63 % |
| 1 | 537 726 976 | 30,729 | 2.93 % |
| 20 | 52 162 560 | 7,631 | 7.49 % |
| 24 | 151 721 296 | 4,506 | 1.52 % |
| 3 | 25 159 442 | 3,926 | 7.99 % |
| 5 | 329 743 512 | 3,675 | 0.57 % |
| 11 | 134 029 577 | 3,374 | 1.29 % |
| 16 | 47 024 784 | 2,578 | 2.81 % |
| 17 | 47 024 784 | 2,578 | 2.81 % |
| 14 | 356 837 650 | 1,999 | 0.29 % |
| 15 | 38 496 260 | 585 | 0.78 % |
| 25 | 117 831 384 | 488 | 0.21 % |
| 8 | 77 248 512 | 264 | 0.17 % |
| 6 | 20 301 246 | 120 | 0.30 % |
| 10 | 60 688 896 | 81 | 0.07 % |
| 26 | 177 439 724 | 4 | 0.00 % |
| 23 | 35 205 120 | 2 | 0.00 % |
| 2 | 2 904 678 | 1 | 0.02 % |
| SUM | | 163,323 | |

***Table 7.9:*** *Number of hits per reference file in the suspect,s unallocated area. Sorted by hits and block-size of 512 bytes.*

### 7.3.3 Detecting hash collision in blocks from unallocated areas

Another testing was to determine SHA256 collisions in the tables *blockhash_unallocated512* and *blockhash_unallocated4096*. This is blocks from unallocated areas on the suspects system volume in the case data.

For the *blockhash_unallocated4096*, 1,491,675 collisions was detected. By exporting the performed query and creating a separate table of these hits and performing a cross query against the *caselookup4096* table, 524 collisions was detected.

There were 8 collisions for file 3, 2 for file 20, 2 for file 26 and 512 for file 13. For file 20 and 26 the entropy was 0.0 while file 3 and 13 generally have high entropy.

| Filenum | Size | Hits | Percent |
|---:|---:|---:|---:|
| 13 | 75 805 306 | 7,191 | 38,9 % |
| 9 | 230 114 636 | 5,406 | 9,6 % |
| 1 | 537 726 976 | 3,824 | 2,9 % |
| 24 | 151 721 296 | 564 | 1,5 % |
| 3 | 25 159 442 | 516 | 8,4 % |
| 11 | 134 029 577 | 422 | 1,3 % |
| 16 | 47 024 784 | 317 | 2,8 % |
| 17 | 47 024 784 | 317 | 2,8 % |
| 20 | 52 162 560 | 236 | 1,9 % |
| 15 | 38 496 260 | 34 | 0,4 % |
| 8 | 77 248 512 | 33 | 0,2 % |
| 6 | 20 301 246 | 15 | 0,3 % |
| 26 | 177 439 724 | 2 | 0,0 % |
| SUM | | 18,877 | |

***Table 7.10:*** *Number of hits per reference file in the suspects unallocated area. Sorted by hits and block-size of 4 096 bytes.*

For file 13 the collisions were two contiguous areas from the reference file, block 7488-7615 (128 blocks / 512 KiB) and 8000-8127 (128 blocks / 512 KiB). A normal assumption of the collisions for file 13 is that the actual reference file could have remained on two different locations of the suspect's storage at a certain time. The mutual blocks, entropy spans from 0.65 to 0.99 and is from medium to very high entropy.

For file 3 the collisions were a contiguous chain of blocks in the reference file. Block 252-255 and the same blocks were found contiguously in the unallocated area in blocks 28,574,892 - 28,574,895 with entropy for all of them at 0.986.

### 7.3.4 Analysing hits between reference data and unallocated area

Further, to visualize the hits between blocks in the reference material and the blocks in unallocated clusters on suspect's storage, a block map of some of the findings is presented. There are two types of block map. One that shows what blocks in the reference data per file is found in unallocated. This block map shows each block starting from 0 to the last block plotted by corresponding pixel in the block map. Each pixel is also coloured to visualize the entropy of that block.

The other type of block map shows where in the unallocated area of the suspect's storage there is found identical blocks from the reference material.

#### 7.3.4.1 Reference data files, 4,096 byte blocks

In this section we look into the 4,096 bytes common blocks from the reference data files. For each file we find common blocks, we have created a block map to illustrate where in the reference data we have common blocks from. The file numbers we look into is the files 1,9,13,20 and 24, 4,096 byte block size. The file numbers is the same as referred to in Table 7.6. All the block maps are created diagrammatically using Python and the content from the database records of common blocks between reference data and unallocated areas.

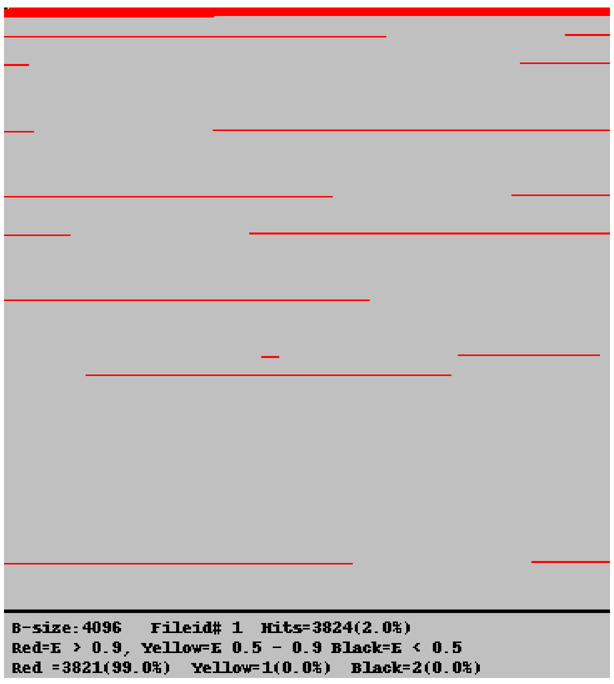
The grey area, is blocks from the reference file that are not located in the unallocated area of the suspect's disk. Red pixels are blocks with high entropy ($> 0.9$) that is found in the unallocated area. Yellow blocks are blocks with medium entropy ($0.5 \leq entropy \leq 0.9$) and black pixels are blocks with low entropy ($< 0.5$).

**Common blocks from file number 1**
The block map in Figure 7.5 shows which blocks in the reference data for file 1 that are located in the unallocated clusters from the suspect's storage. File 1 is about 537 MB. The pixel in upper left corner is the first block of the reference file and the bottom right is the last block from the same file. Each pixel represents a block from the file in sequential order.

The block map shows that almost all blocks located between reference and unallocated area have high entropy and 3,824 blocks are identified which is 2.0 % of the total number of blocks the reference file occupy (see Table 7.6). We notice, many of the findings are sequential areas of several hundreds of blocks, each of 4,096 bytes.

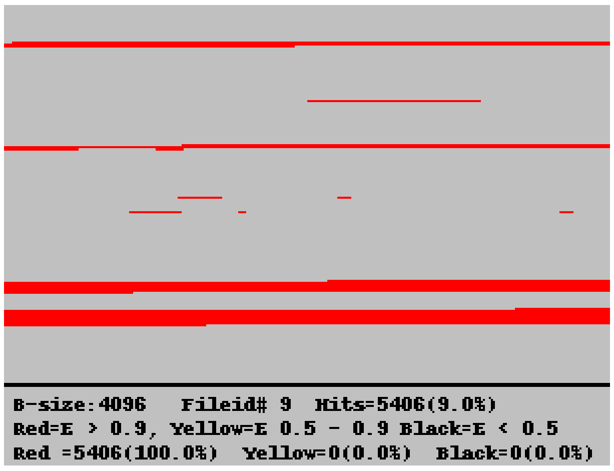
**Common blocks from file number 9**
The block map in Figure 7.6 shows much the same as Figure 7.5. This is a smaller reference file (about 230 MB, less than halv of file 1) and there are 5.406 (9.0%) mutual blocks in the reference file and unallocated area. As we can observe, all common blocks have a high entropy. We also notice that the common blocks referred to from this file are several large contiguous chains. Two of the chains are severe.

**Common blocks from file number 13**
The block map in Figure 7.7 shows a high amount of common blocks between the reference data and the unallocated clusters. in Table 7.6 this reference file is about 75 MB ( 18,507 4 KiB blocks) and has 7,191 in common (38.0%).



```
B-size:4096   Fileid# 1  Hits=3824(2.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =3821(99.0%)  Yellow=1(0.0%)  Black=2(0.0%)
```

***Figure 7.5:*** *Block map of reference object 1, block size 4096*



```
B-size:4096   Fileid# 9  Hits=5406(9.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =5406(100.0%)  Yellow=0(0.0%)  Black=0(0.0%)
```

***Figure 7.6:*** *Block map of reference object 9, block size 4096*

Another observation here is the high amount of blocks (87% of the common blocks) with medium entropy (yellow pixels) but also a relatively high number (12% of the common blocks) of blocks with high entropy. From the block map we can identify more than 4 major contiguous areas of common blocks.

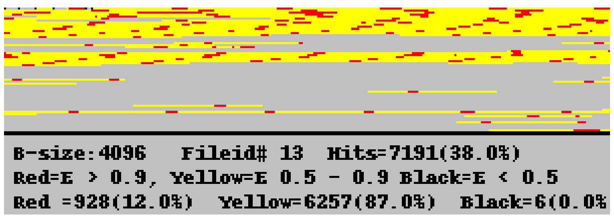


***Figure 7.7:*** *Block map of reference object 13, block size 4096*

**Common blocks from file number 20**

The block map in Figure 7.8 is a block map from file 20 from Table 7.6 of 52 MB. In the unallocated area of suspects storage, there is 236 common blocks of 4 KiB which represents about 1.8%. Most of the common blocks have either high or medium entropy.

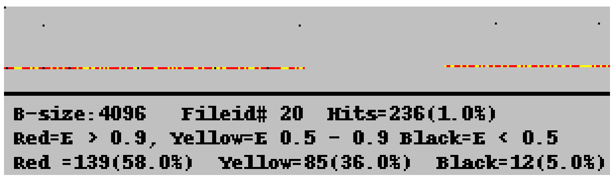


***Figure 7.8:*** *Block map of reference object 20, block size 4096*

**Common blocks from file number 24**

The block map in Figure 7.9 is a block map from file 24 from Table 7.6 of 151 MB. In the unallocated area of suspects storage, there is 564 common blocks of 4 KiB which represents about 1.5%. All common blocks except two have high entropy.

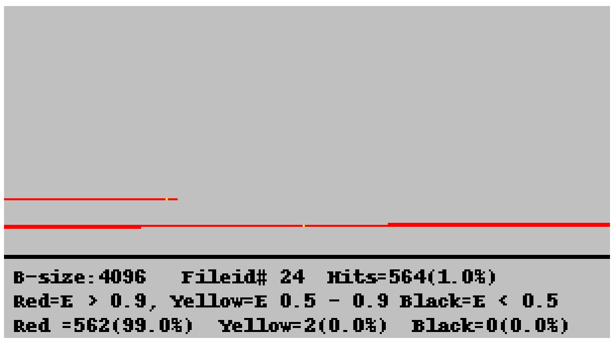


***Figure 7.9:*** *Block map of reference object 24, block size 4096*

#### 7.3.4.2 Reference data files, 512 byte blocks

In this section we look into the 512 bytes common blocks from the reference data files. For each file we find common blocks, we have created a block map to illustrate where in the reference data we have common blocks from. The file numbers we look into is the files 1,9,13,20 and 24, 4,096 byte block size. How the maps are created and the explanation of colors and symbols are the same as in Section 7.3.4.1. All file numbers refers to the file Table 7.6.

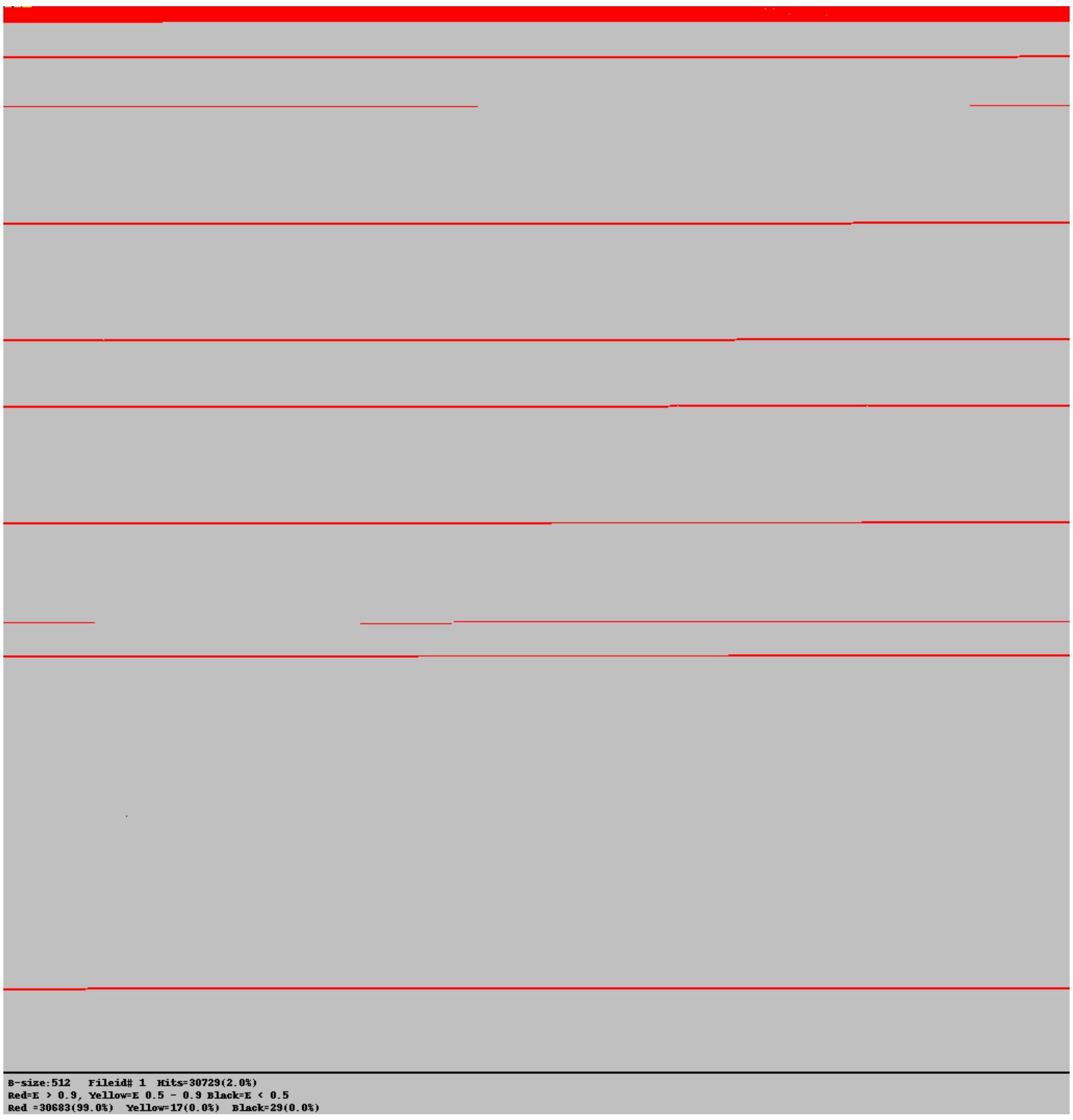
**Common blocks from file number 1** . The block map in Figure 7.10 is from file 1 in the file Table 7.6. There are several contiguous areas of common blocks between the reference file and unallocated clusters. The reference file is 537 MB and there are 30,729 common blocks. 99.0% of these common blocks have high entropy. The rest are just a few blocks, mainly from the beginning of the file.

**Common blocks from file number 9**
The block map in Figure 7.11 is from file 9 in the file Table 7.6. There are several contiguous areas of common blocks between the reference file and unallocated clusters. The reference file is 230 MB and there are 43,274 common blocks. 99.0% of these common blocks have high entropy. The blocks with low entropy are mainly from the beginning of the file.

**Common blocks from file number 13**
The block map in Figure 7.12 is from file 13 in the file Table 7.6. There are several contiguous areas of common blocks between the reference file and unallocated clusters. The reference file is 75 MB and there are 57,508 common blocks. 87.0% of these common blocks have medium entropy. There are also a significant number of blocks with high entropy, 7,332 blocks. The blocks with low entropy are mainly from the beginning of the file.

**Common blocks from file number 20**
The block map in Figure 7.13 is from file 20 in the file Table 7.6. There is just one large contiguous area of common blocks between the reference file and unallocated clusters. The rest of the hits are very small areas with just a few contiguous blocks. The reference file is 52 MB and there are 7,631 common blocks.

The hits are spread throughout the file and the only large contiguous area has high entropy.



```
B-size:512   Fileid# 1  Hits=30729(2.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =30683(99.0%)  Yellow=17(0.0%)  Black=29(0.0%)
```

***Figure 7.10:*** *Block map of reference object 1, block size 512*

**Common blocks from file number 24**

The block map in Figure 7.14 is from file 24 in the file Table 7.6. There are two relatively large contiguous areas of common blocks between the reference file and unallocated clusters. The reference file is 151 MB and there are 4,506 common blocks.

The two common areas are mainly blocks with high entropy, about 99% of the common blocks.

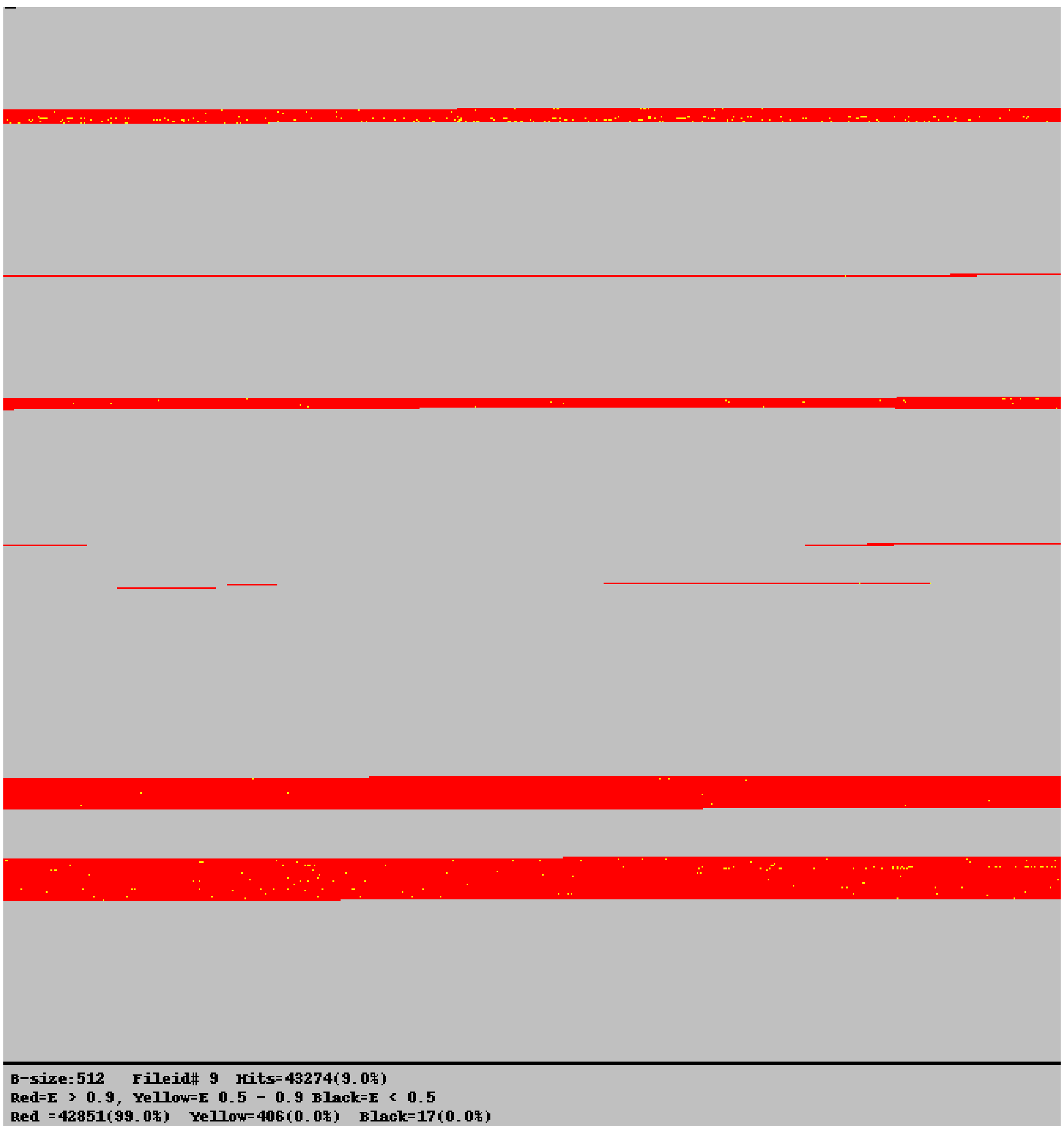
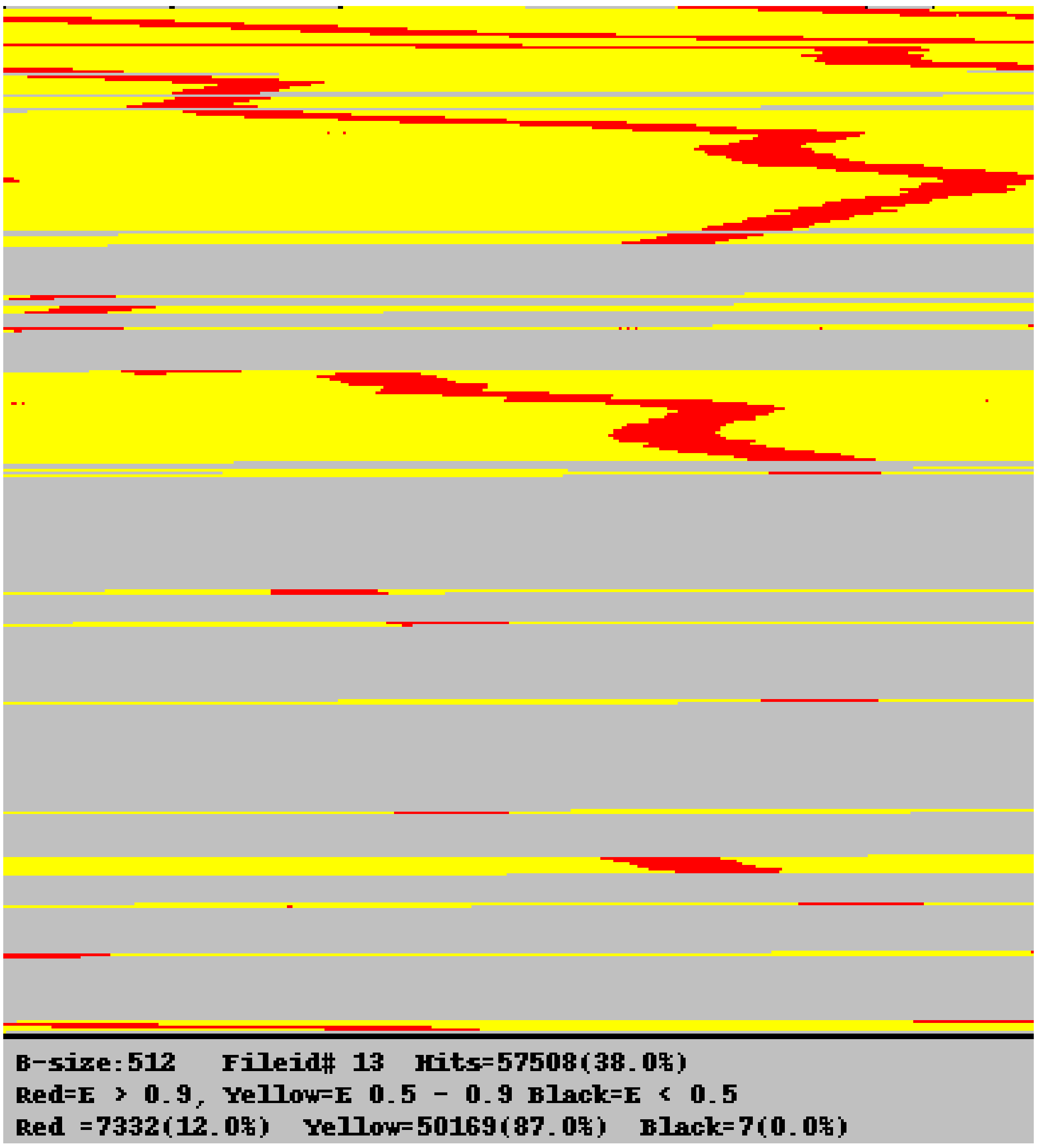


***Figure 7.11:*** *Block map of reference object 9, block size 512*

### 7.3.5 Connection between common blocks in unallocated areas and reference files

In the previous sub-chapters we have looked into some of the reference files and what blocks per file are found in unallocated area of suspect's storage. The reference to file number from reference files, in the file Table 7.6. In this section, we will visualize the placement of these hits in the unallocated area per file. There is one block map per file. Each map is from the first common block to the last common block. Blocks located in unallocated areas are marked with a dashed green border on the major areas to increase the visibility.



```
B-size:512   Fileid# 13  Hits=57508(38.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =7332(12.0%)  Yellow=50169(87.0%)  Black=7(0.0%)
```

***Figure 7.12:*** *Block map of reference object 13, block size 512*

The unallocated area from the suspect's storage is not a contiguous area but a collection of single blocks and range of blocks. However, the unallocated blocks is in sequential order as they are present on the storage.

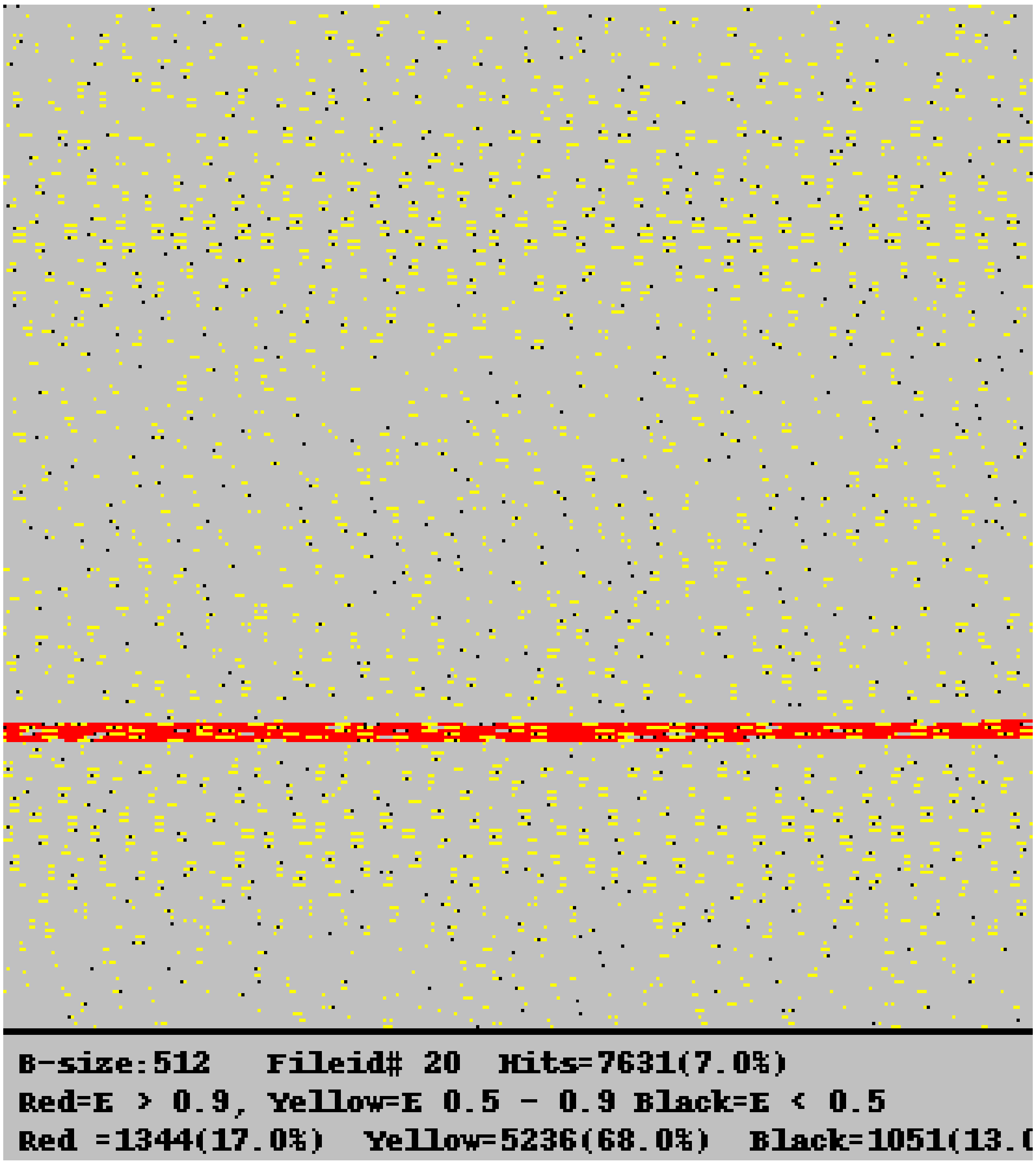
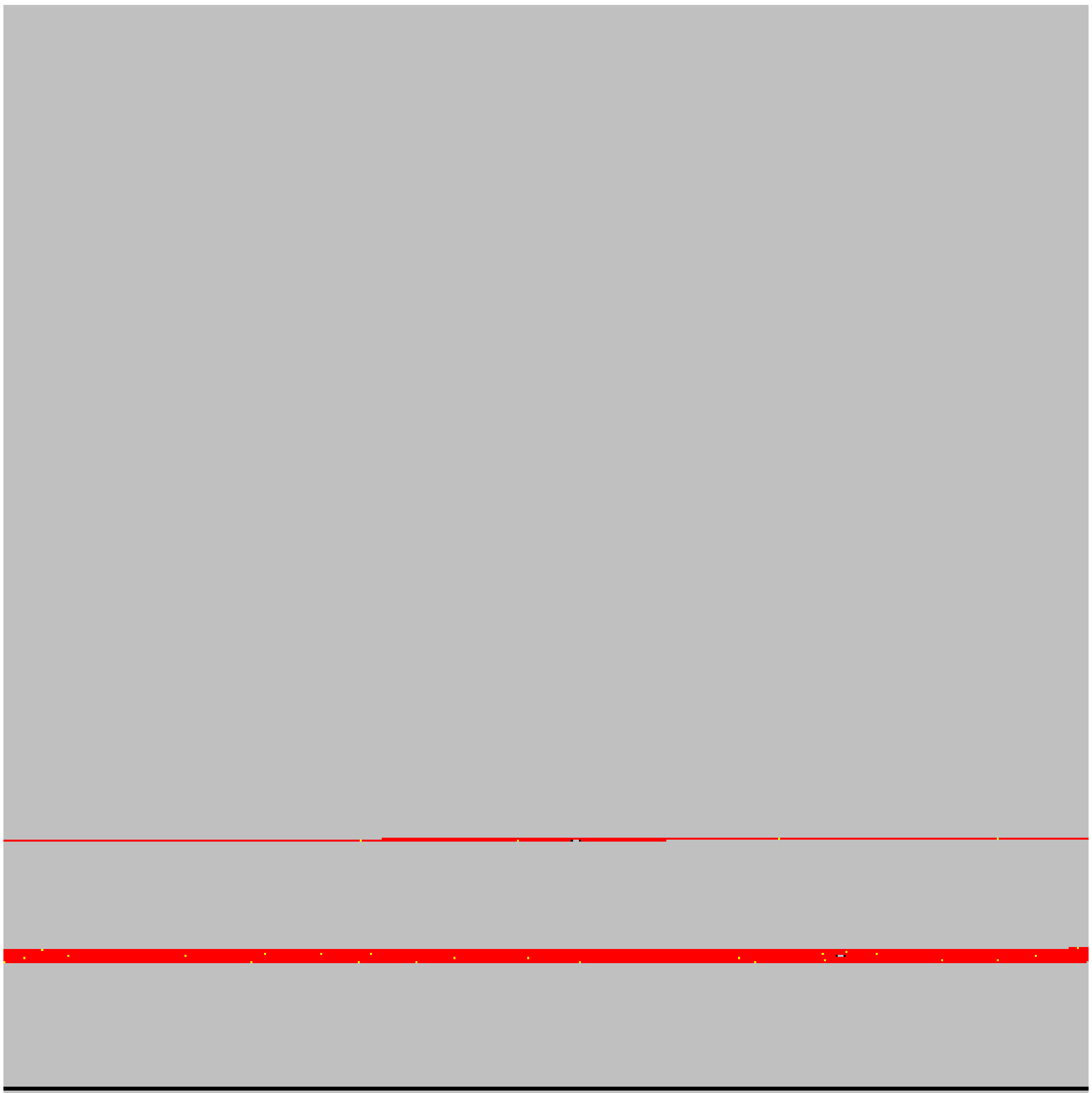


*Figure 7.13: Block map of reference object 20, block size 512*

The first unallocated block is block 0. We are using 512 and 4,096 byte blocks. The suspect's storage make use of 4,096 byte blocks (clusters) in the NTFS file system. The 512 byte block hashes of the unallocated area spans from block 0 to 290,646,016 and in 4 KiB blocks, the block numbers span from 0 to 36,330,752



```
B-size:512   Fileid# 24  Hits=4506(1.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =4474(99.0%)  Yellow=28(0.0%)  Black=4(0.0%)
```

***Figure 7.14:*** *Block map of reference object 24, block size 512*

On each block map from unallocated area, there is a legend at the bottom. The value "Range u.b" is the placement in the unallocated area using "B-size" as the unit for block-size. In maps not using 512 byte blocks, the "Range u.b" is converted into 512 byte blocks numbering. This just to have a common method of numbering blocks between maps related to the same file.

The legend value ”Res.” is the amount of blocks per pixel (resolution) in the block map. A value of 100 means each pixel is 100 blocks.

Most of the blocks from the reference file found in unallocated areas in the unallocated map are highlighted with a green dashed line. Some small block areas are not highlighted.

#### 7.3.5.1 Common blocks in unallocated areas, 512 byte blocks

In this section we will look into the common blocks in unallocated areas for block size 512 bytes. The following section will do the same on 4,096 byte blocks.

**Common blocks in unallocated areas from file number 1**
The block map in Figure 7.15 show where the common blocks from file 1 remain in the unallocated area. There are two large areas, block 14,158,688 - 14,702,855 (11,096 blocks and 232,312,061 - 234,531,071 (19,539 blocks). These two areas are not completely contiguous but remain in the same area. Between those two areas, there are 94 blocks in several small chunks.

**Common blocks in unallocated areas from file number 9**
The block map in Figure 7.16 shows where the common blocks from file 9 remains in the unallocated area. There are several dominating range of blocks. Block range 42,639,576 - 43,279,239 (472 blocks), 68,347,745 - 68,347,751 (7 blocks) and 111,272,152 - 111,947,751 (42,785 blocks).

**Common blocks in unallocated areas from file number 13**
The block map in Figure 7.17 shows where the common blocks from file 13 remains in the unallocated area. There is one large range of blocks, 104,656,944-121,028,583 (56,820 blocks). This is the marked area a little above the middle of the block map. In addition, there are several minor areas of common blocks, 20,744,256 - 22,125,063 (200 blocks), 35,786,376 - 37,355,863 (232 blocks), 66,695,664-66,899,055 (128 blocks), 232,311,280-232,761,487 (128 blocks).

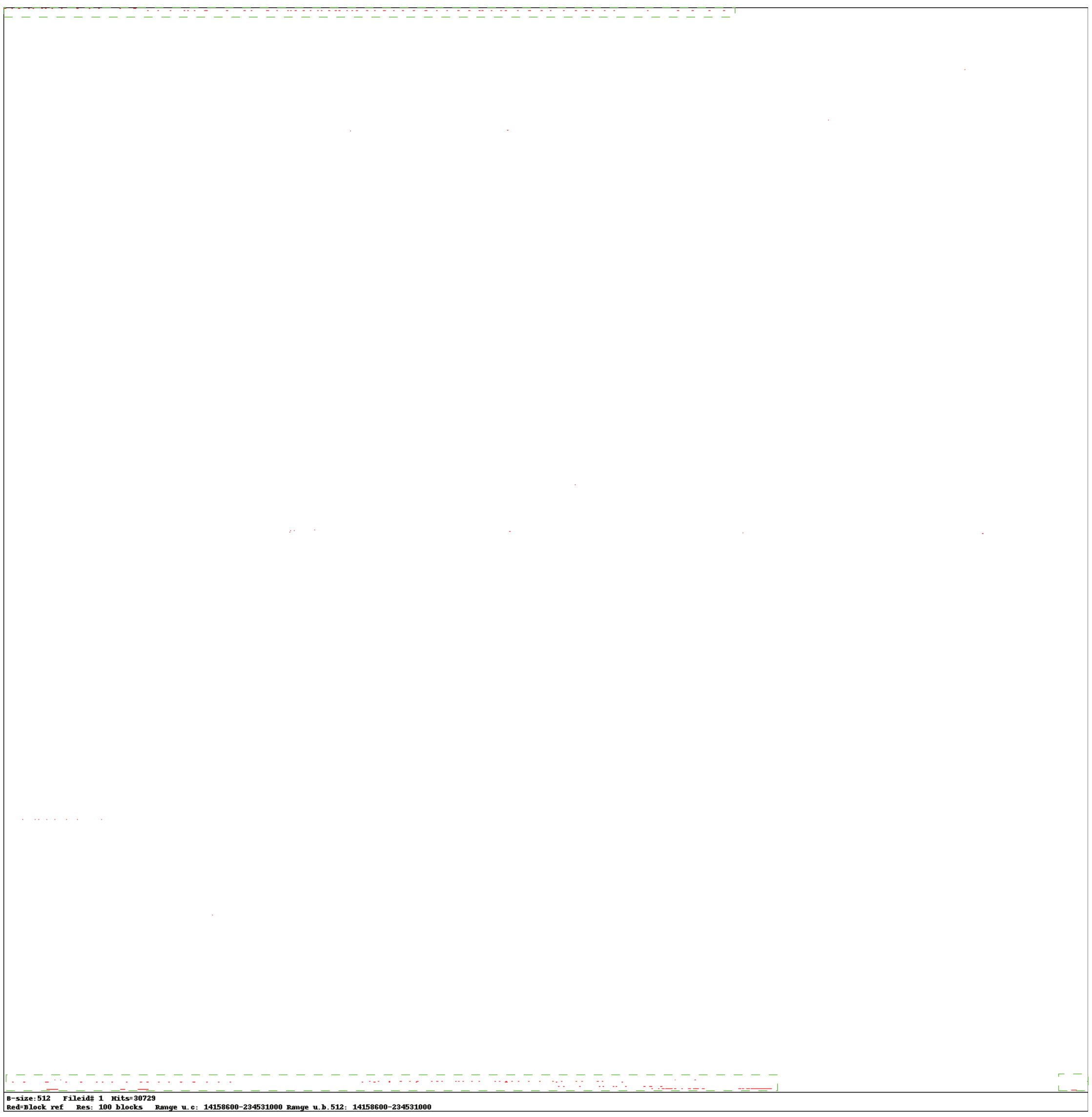


***Figure 7.15:*** *Block map of placement in unallocated areas object 1, block size 512*

**Common blocks in unallocated areas from file number 20**

The block map in Figure 7.18 shows where the common blocks from file 20 remains in the unallocated area. This file has just a few minor block areas in the unallocated area. Block 56,992,508 (The file hader block), 80,273,788 and 81,942,750 (5 blocks), block 93,220,055, 111,064,319 - 120,643,116 (7 blocks), 166,073,928 - 166,075,775 (1,742 blocks), block 198,456,951, 211,586,841 - 211,590,507 (495 blocks) and 232,661,984 - 232,661,988 ( 2 blocks).

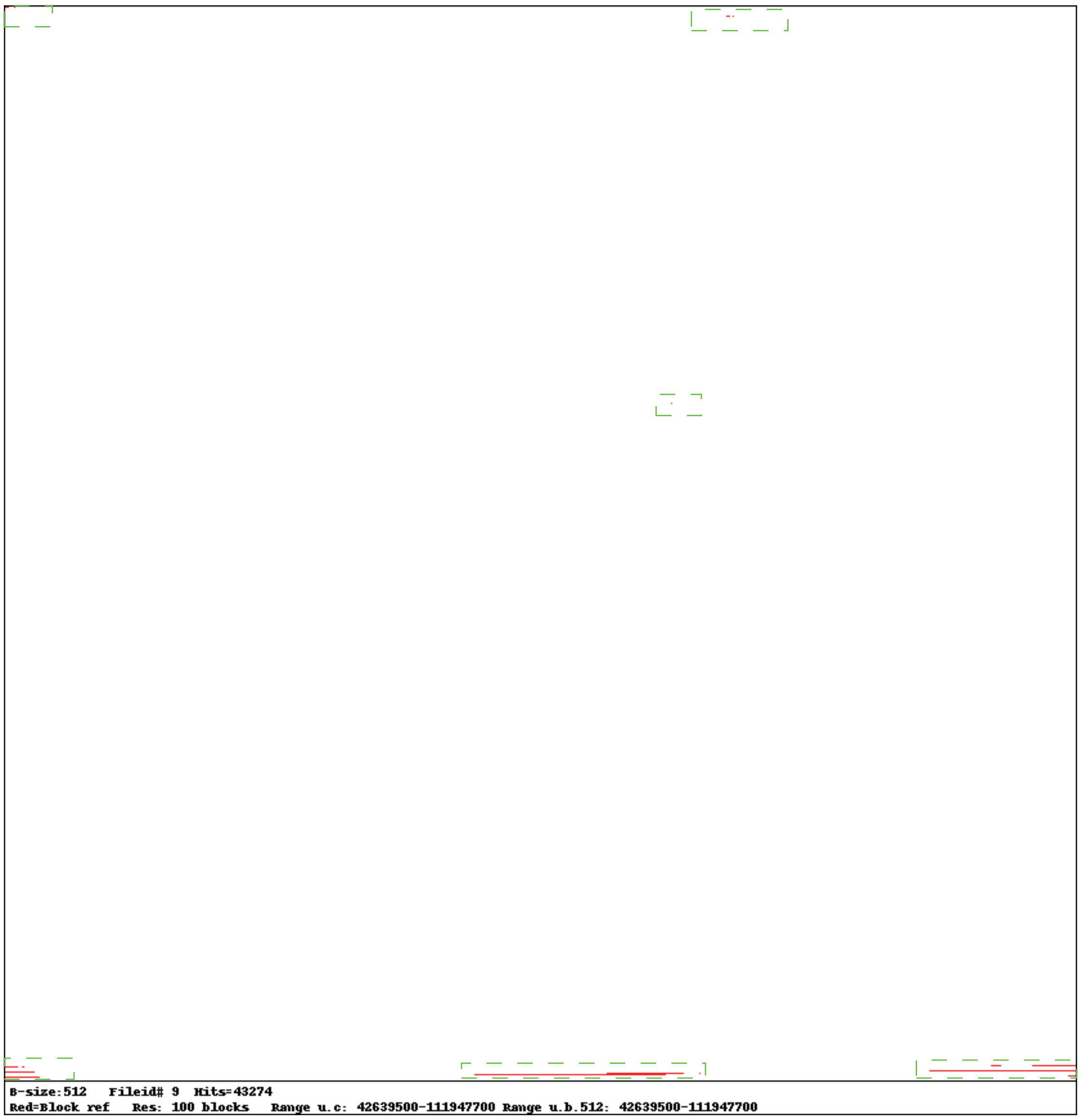


***Figure 7.16:*** *Block map of placement in unallocated areas object 9, block size 512*

**Common blocks in unallocated areas from file number 24**

The block map in Figure 7.19 show where the common blocks from file 24 remains in the unallocated area. This file has two block areas in the unallocated area. Block areas 59,397,272 - 59,401,095 (3,821 blocks), 105,930,280 - 105930967 (685 blocks).

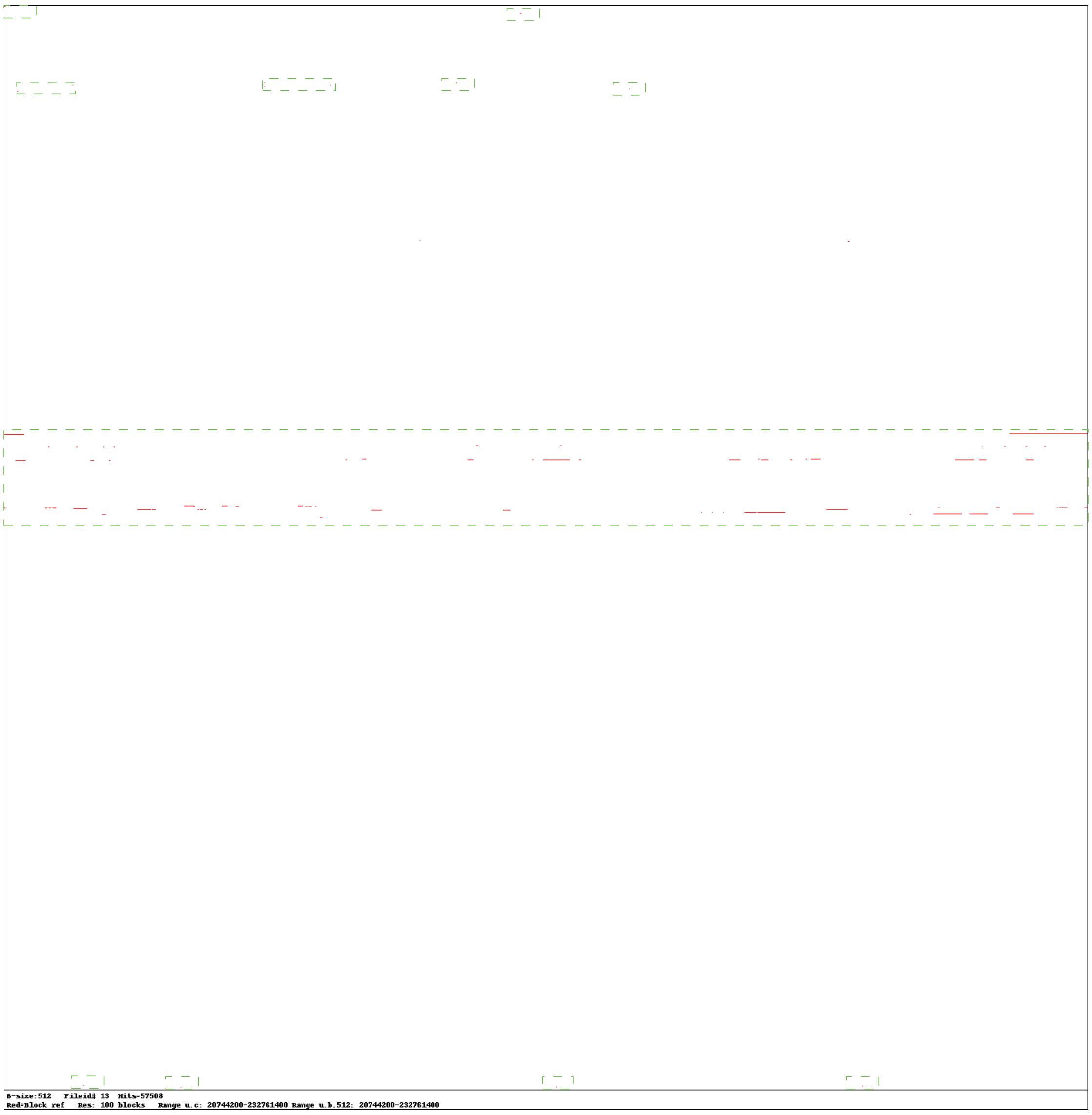


***Figure 7.17:*** *Block map of placement in unallocated areas object 13, block size 512*

#### 7.3.5.2 Common blocks in unallocated areas, 4,096 byte blocks

In this section we will look into the common blocks in unallocated areas for block size 4,096 bytes.

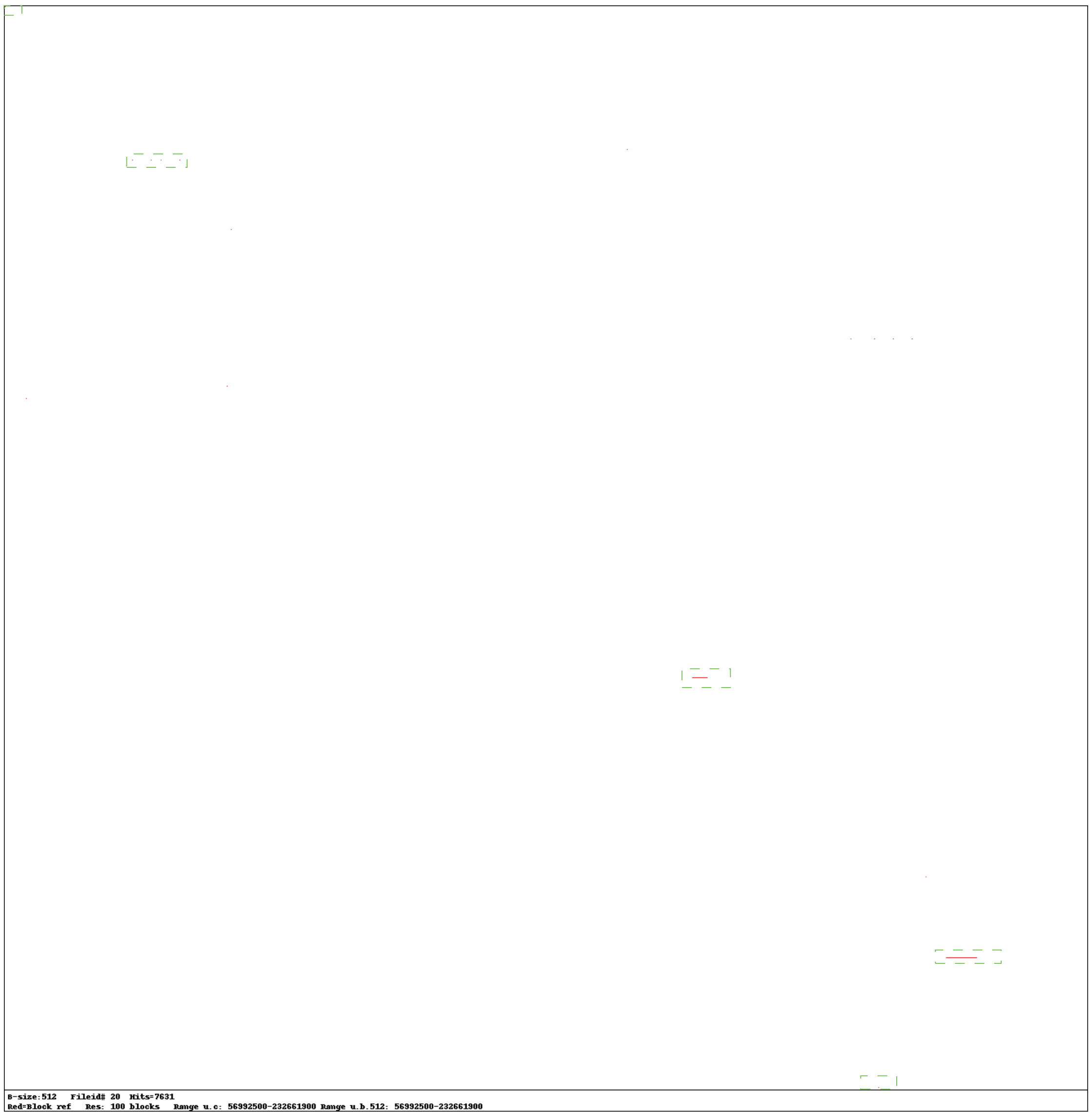


***Figure 7.18:*** *Block map of placement in unallocated areas object 20, block size 512*

## Common blocks in unallocated areas from file number 1

The block map in Figure 7.20 shows where the common blocks from file 1 remain in the unallocated area.

There are three areas of blocks, block 1,769,836 - 1,772,075 (256 blocks), 1,827,969 - 1,837,856 (1,131 blocks), 29,081,520 - 29,316,383 (2,437 blocks). These three areas are not completely contiguous but remain in the same area.

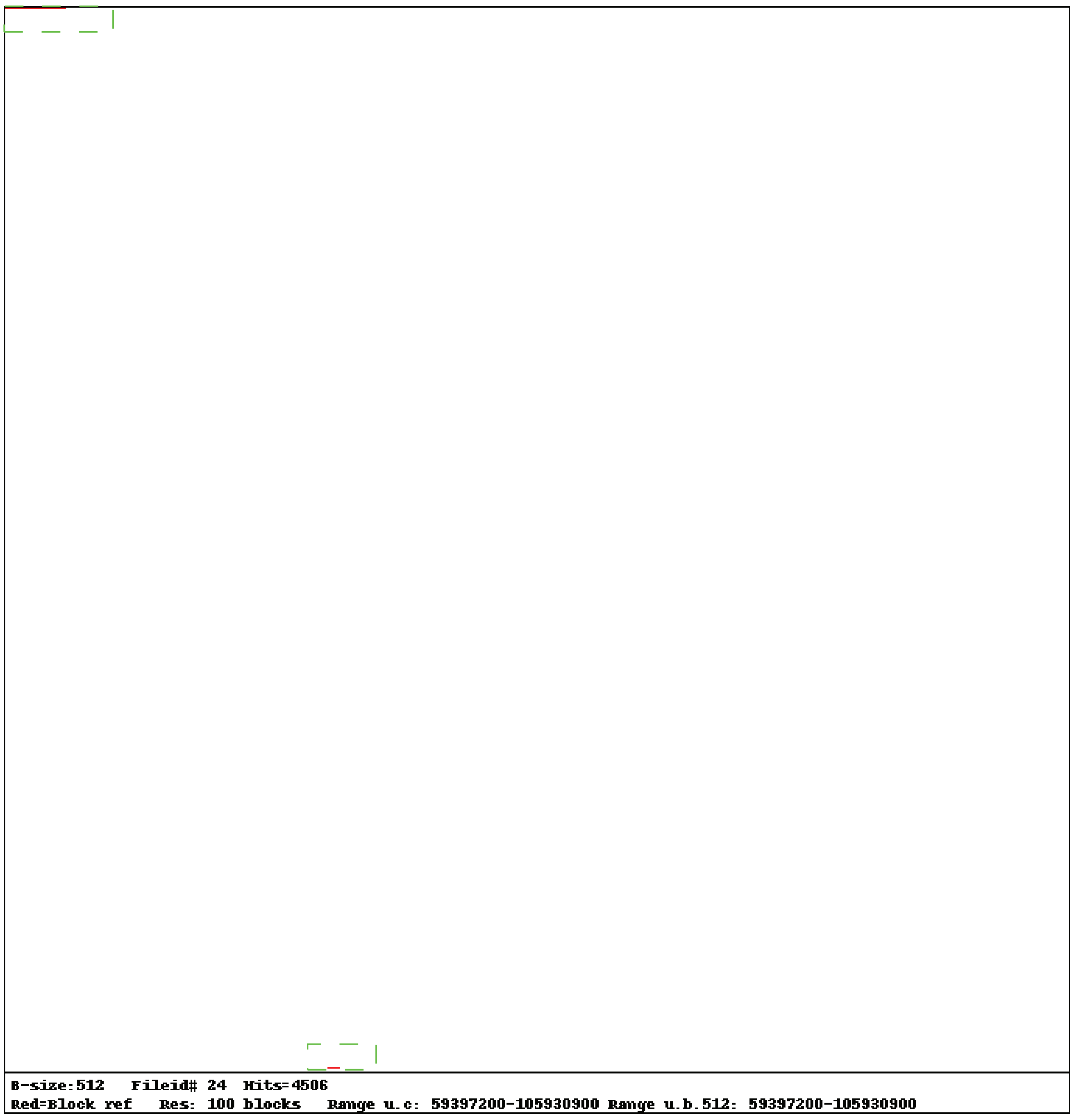


***Figure 7.19:*** *Block map of placement in unallocated areas object 24, block size 512*

**Common blocks in unallocated areas from file number 9**

The block map in Figure 7.21 shows where the common blocks from file 9 remain in the unallocated area.

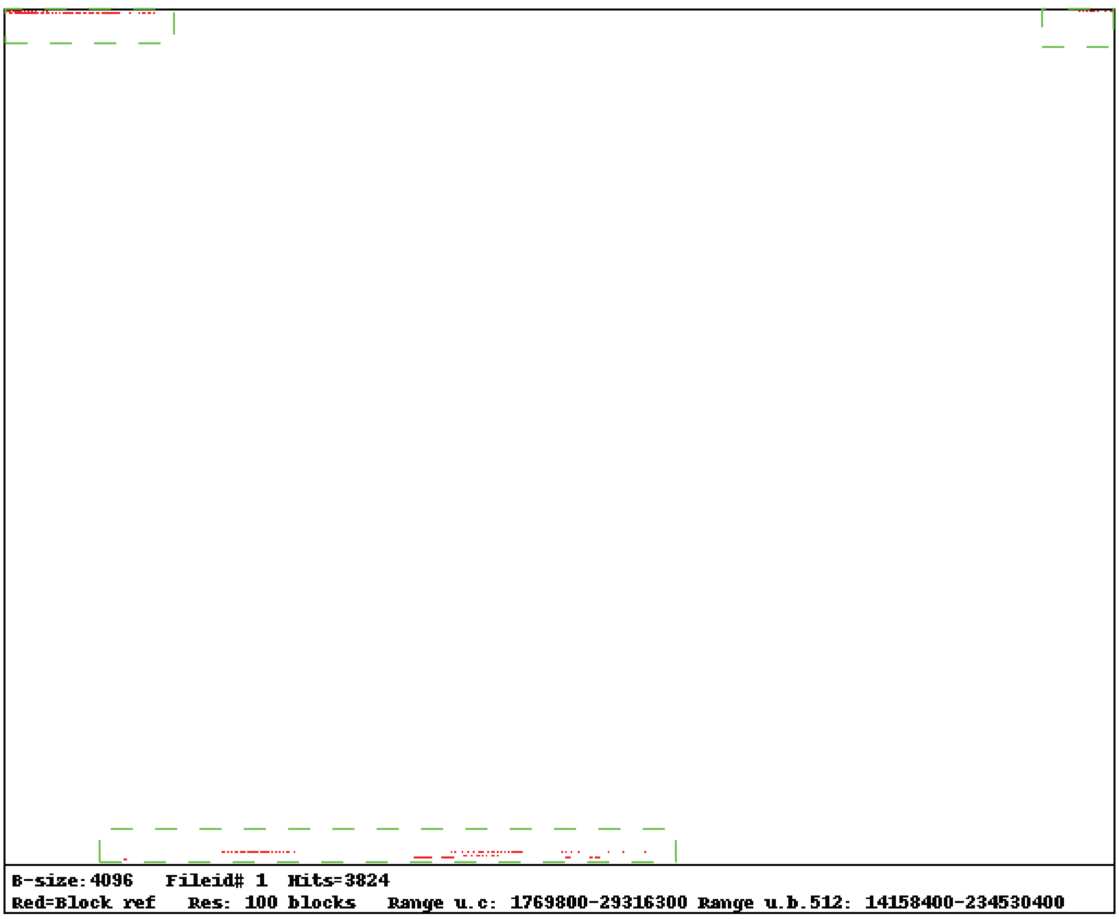


***Figure 7.20:*** *Block map of placement in unallocated areas object 1, block size 4096*

There are two areas of blocks, block 5,329,947 - 5,409,904 (59 blocks), 13,909,019 - 13,993,468 (5,347 blocks). These areas are not completely contiguous but remain in the same area.

**Common blocks in unallocated areas from file number 13**

The block map in Figure 7.22 shows where the common blocks from file 13 remain in the unallocated area.

There are five areas of blocks, block 2,593,032 - 2,765,632 (25 blocks), 4,473,297 - 4,669,482 (29 blocks), 8,336,958 - 8,362,381 (16 blocks), 13,082,118 - 15,128,572 (7,105 blocks) and 29,038,910 - 29,095,185 (16 blocks). These areas are not completely contiguous but remain in the same area.

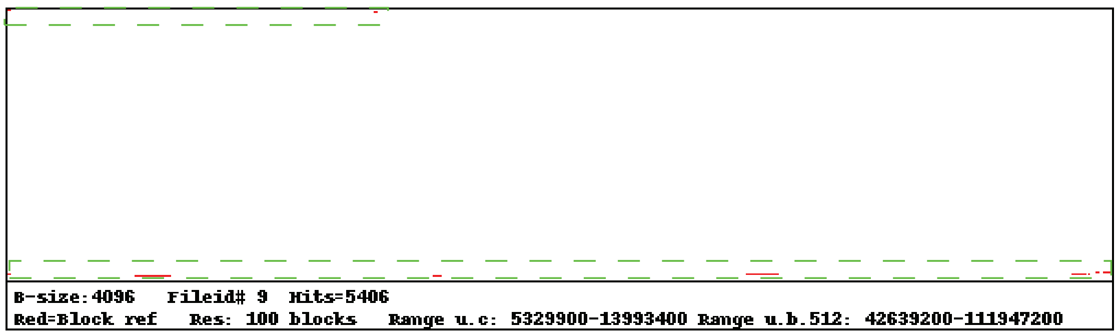


***Figure 7.21:*** *Block map of placement in unallocated areas object 9, block size 4096*

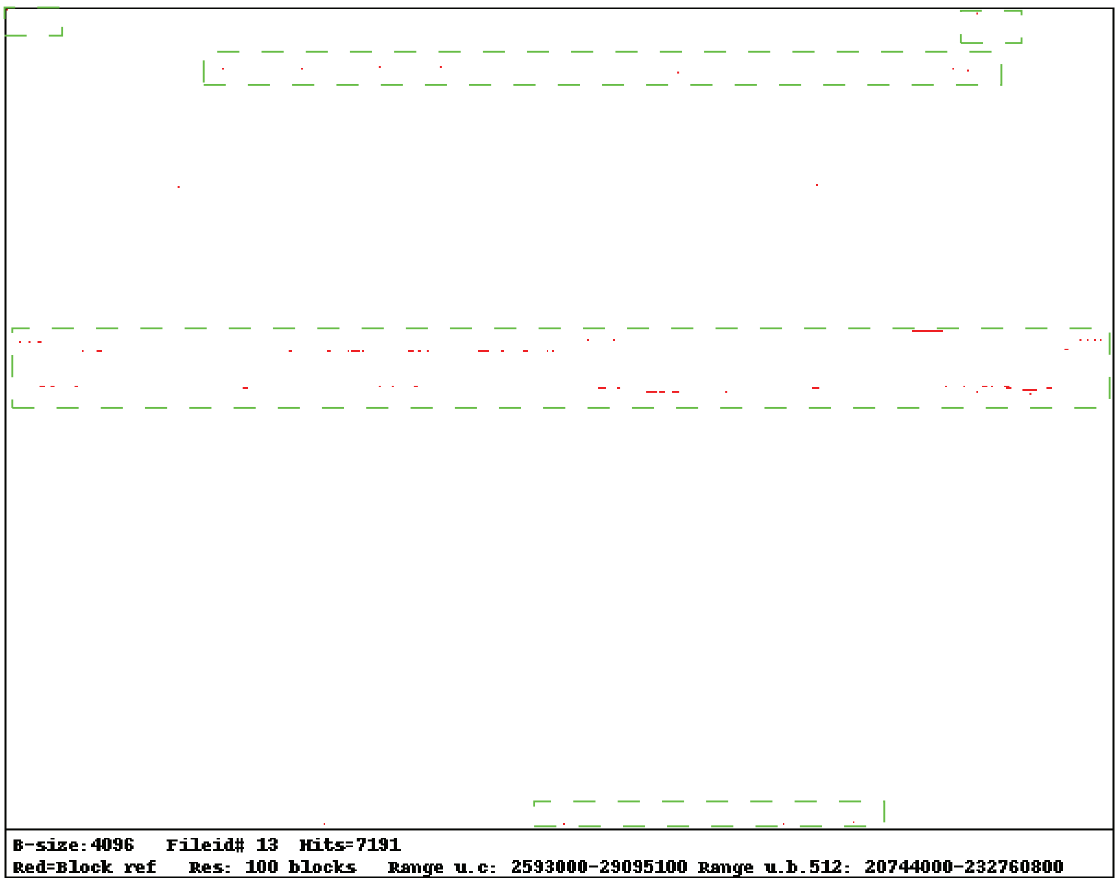
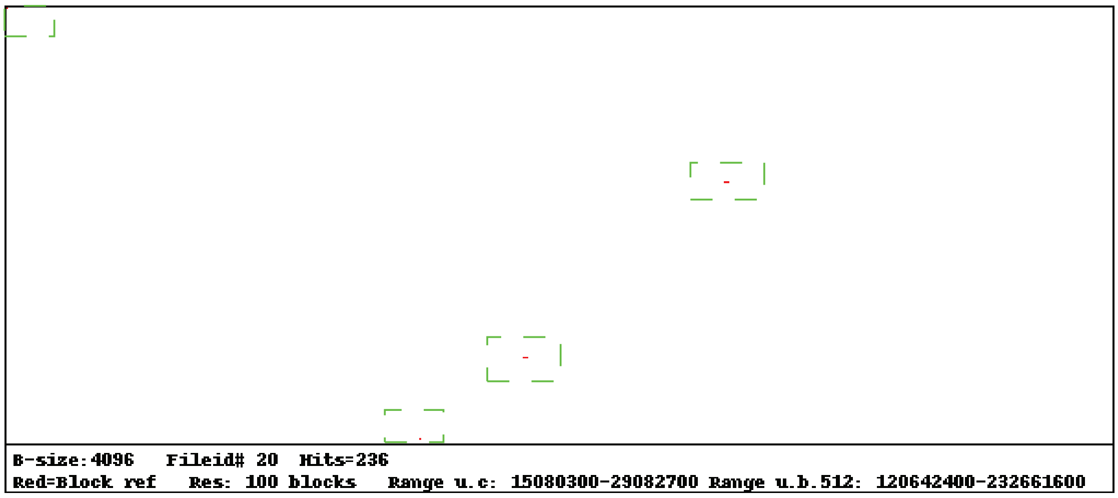


***Figure 7.22:*** *Block map of placement in unallocated areas object 13, block size 4096*

### Common blocks in unallocated areas from file number 20

The block map in Figure 7.23 shows where the common blocks from file 20 remain in the unallocated area.

There are four areas of blocks, the single block 15,080,389, 20,759,241 - 20,759,471 (230 blocks), 26,448,387 - 26,448,590 (4 blocks)and block 29,082,748. Not all of these areas are completely contiguous but remain in the same area.



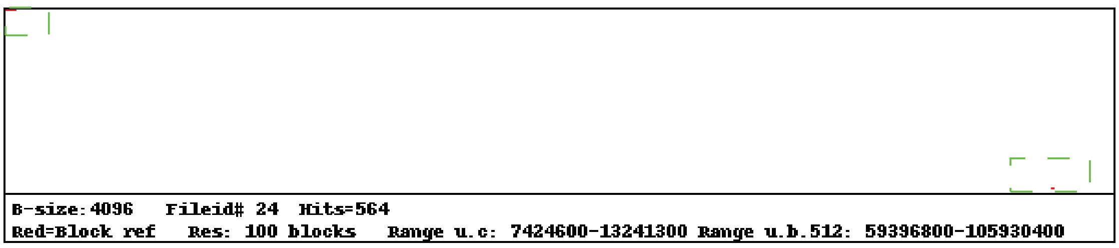
```
B-size:4096   Fileid# 20  Hits=236
Red=Block ref   Res: 100 blocks   Range u.c: 15080300-29082700 Range u.b.512: 120642400-232661600
```

*Figure 7.23: Block map of placement in unallocated areas object 20, block size 4096*

### Common blocks in unallocated areas from file number 24

The block map in Figure 7.24 shows where the common blocks from file 24 remain in the unallocated area.

There are two areas of blocks, 7,424,659 and 7,425,136 (478 blocks) and 13,241,285 and 13,241,370 (86 blocks). Not all of these areas are completely contiguous but remain in the same area.



```
B-size:4096   Fileid# 24  Hits=564
Red=Block ref   Res: 100 blocks   Range u.c: 7424600-13241300 Range u.b.512: 59396800-105930400
```

***Figure 7.24:*** *Block map of placement in unallocated areas object 24, block size 4096*

## 7.4 Results from dataset 4, misc datasets

### 7.4.1 Database msc_veracrypt

This dataset is created from a 25 GB Veracrypt AES encrypted container. The total number of rows in the table containing blockhashes with entropy from blocksizes 512 - 16384 is 103,219,200 rows.

The table with SHA256 hashes was checked for collisions. None of the SHA256 has any collisions.

| Block size | Entropy |
|---:|---:|
| 512 | 0.9488 |
| 1,024 | 0.9760 |
| 2,048 | 0.9885 |
| 4,096 | 0.9943 |
| 8,192 | 0.9972 |
| 16,384 | 0.9986 |

***Table 7.11:*** *Average entropy in veracrypt AES encrypted container*

Figure 7.25 shows the distribution of entropy (Y-axis) per block-size (X-axis). The figure confirms what we have observed in similar tests on dataset one and two. The entropy increases as the block-size increase. The most significant difference is from block-size 512 bytes to 1,024. From block-size 8,192 to 16,383 the increase is significantly lower at it seems like entropy flatten out above 4,096 block-size.

### 7.4.2 Database msc_text

This dataset is generated from text parsed from the forensic image used in database msc_case using the strings command in UNIX (OS X). The table with all block-hashes from 512 to 16384 is 1,040,731 rows.

The dataset was checked for collisions.

Figure 7.26 and Table 7.12 illustrates the entropy distribution (Y-axis) per block-size (X-axis). Generally, the entropy on plain ASCII text is not very high. However, the entropy increases as the block-size increase. The curve is different for the one we have in Figure 7.25. In ASCII text, the curve is more linear and seems to increase severe on all selected block-sizes. We have not tested for larger blocks to eventually find a block-size where the entropy flatten out more. Like the entropy on encrypted blocks from AES encrypted data, the steepest part of the curve is between 512 byte-blocks and 1,024.

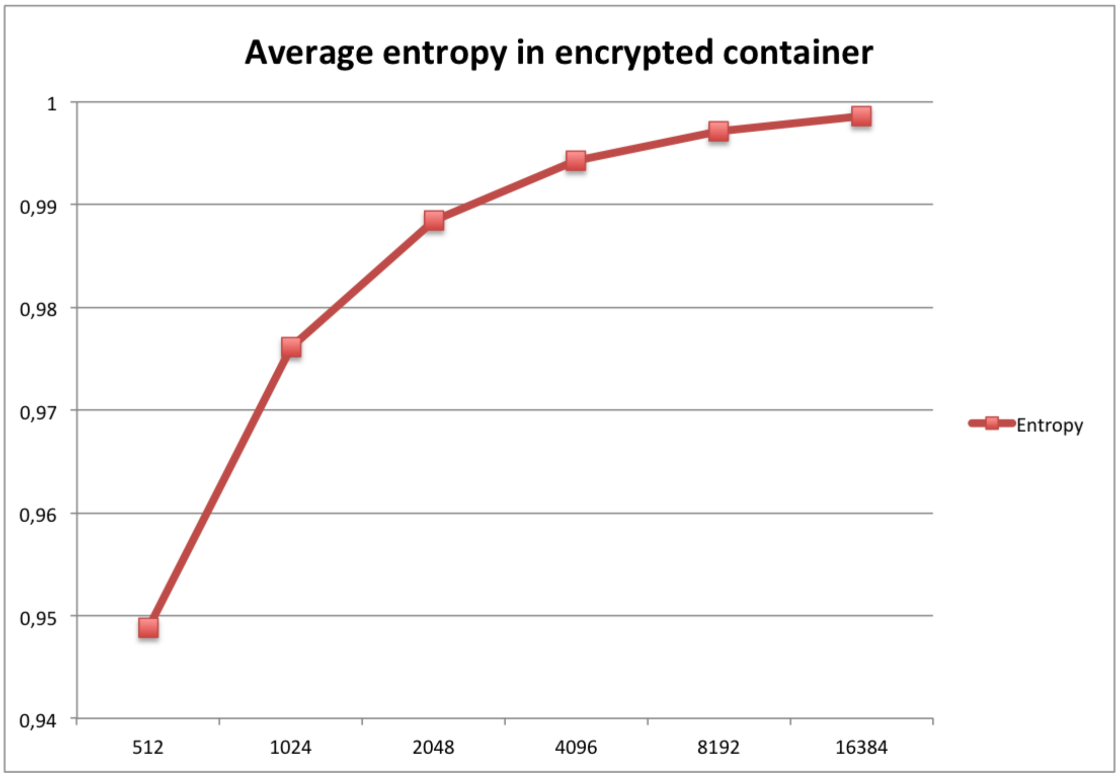


***Figure 7.25:*** *Average entropy in AES encrypted container per block-size.*

| Block size | Entropy |
|---:|---:|
| 512 | 0,6251 |
| 1,024 | 0.6381 |
| 2,048 | 0.6478 |
| 4,096 | 0.6559 |
| 8,192 | 0.6633 |
| 16,384 | 0.6703 |

***Table 7.12:*** *Average entropy in ASCII text*

The dataset with blocks from ASCII text was also tested for SHA256 collisions. This is illustrated in Table 7.13 and Figure 7.27 that represents the table. The figure uses different block-sizes from 512 to 16,384 bytes (X-axis). There are two Y-axis. The left one is number of SHA256 collisions and the right one is duplicate blocks. Both the red and blue graph is very significant. By increasing the block-size the number of duplicate blocks are reduced. As observed in other dataset, the most significant difference is between block-size 512 and 1,024 bytes. When we reach block-size 4,096 bytes, the curve flatten out.

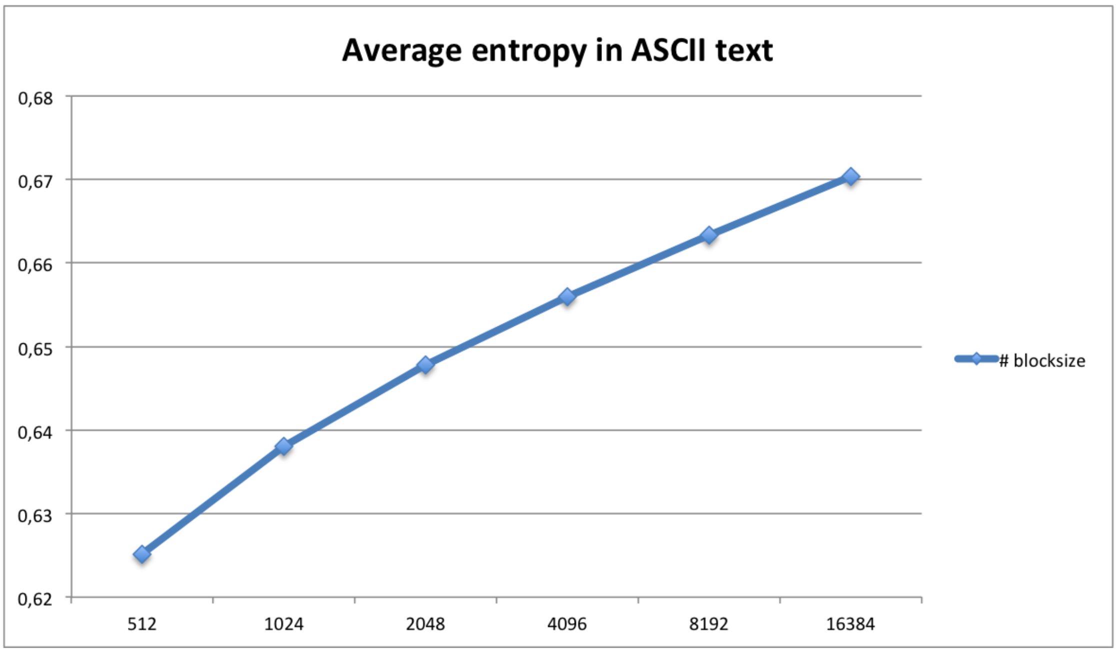


***Figure 7.26:*** *Average entropy in ASCII text per block-size.*

| Block size | SHA256 hits | Duplicate blocks | Max entropy |
|---:|---:|---:|---:|
| 512 | 2,367 | 25,063 | 0.745 |
| 1,024 | 608 | 10,141 | 0.770 |
| 2,048 | 77 | 4,233 | 0.737 |
| 4,096 | 38 | 1,878 | 0.157 |
| 8,192 | 23 | 761 | 0.156 |
| 16,384 | 7 | 290 | 0.000 |
| SUM | 3,120 | 42,366 | |

***Table 7.13:*** *Collisions on different block sizes in msc_text database.*

The two graphs are to some extent coherent. The "SHA256 collisions" are unique SHA256 hashes involved in block collisions (duplicate blocks) while "duplicate blocks" are total number of blocks involved in the collisions. On block size 512 bytes, the average number of block collisions per SHA256 hash is $\frac{26{,}063}{2{,}367} \approx 10$.

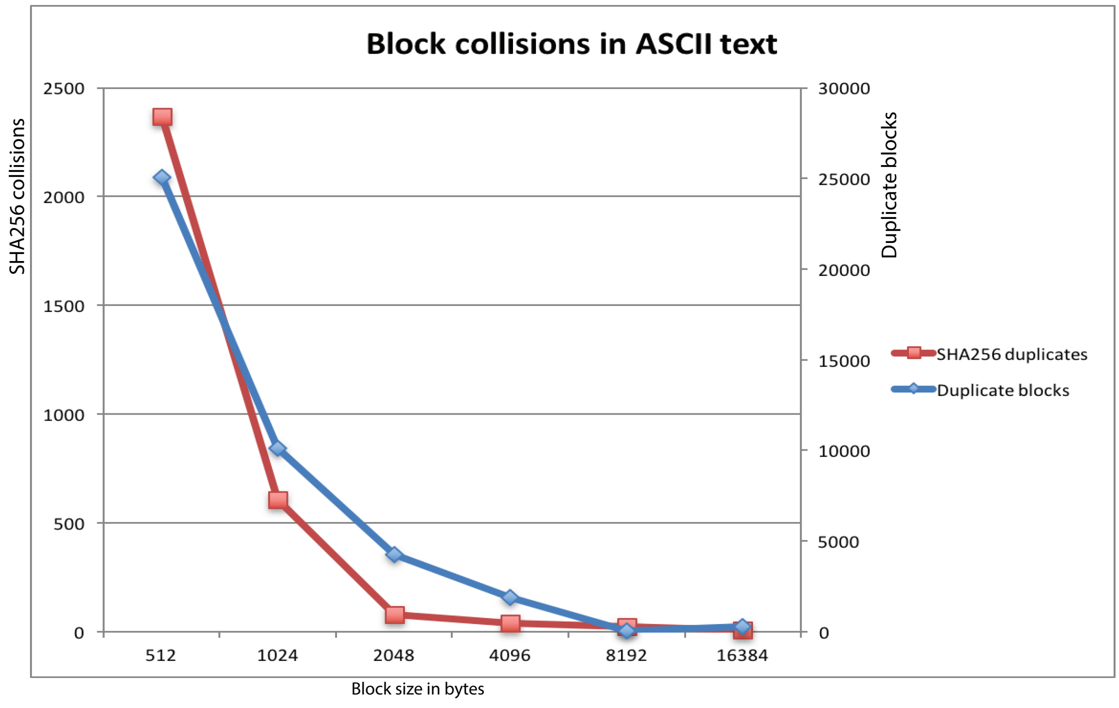


*Figure 7.27: SHA256 hash collisions and duplicate blocks in ASCII text per block-size.*

In Table 7.13 we have the column "Max entropy". This is the highest entropy found among the SHA256 collisions. We notice that the highest entropy on block-size 512 to 2,048 bytes is between 0.745 to 0.737. There is a significant drop in entropy when the blocks size increase to 4,096 bytes which is an indication of low probability of hash collisions on large blocks with relatively high entropy.

### 7.4.3 Entropy in different file types

| Filetype | Average entropy | Number of files | Size of files in MB |
|---|---|---|---|
| ZIP | 0.99 | 934 | 15,900 |
| PY | 0.57 | 7,130 | 89 |
| PDF | 0.90 | 13,415 | 17,043 |
| DOCX | 0.96 | 702 | 738 |
| TXT | 0.56 | 700 | 8,171 |

*Table 7.14: Average entropy on different file types.*

Table 7.14 is a minor test on entropy in different file types other than the one use in previous datasets. All files are picked from a running computer with all kind of file types. The column "Filetype" is different file types identified by file-suffix. The third column is the number of such files and the fourth column is the summary og MB these files occupy. As an example, we have 13,415 PDF files with total of 17 GB. The second column, "Average entropy", is the average entropy per file type. Typically text files like TXT and PY (Python source files) has a generally medium entropy while file formats using different types of compression have generally a high entropy (0.9 and above).

# Part IV

# Discussion and Conclusion

# 8 Discussion

The use of hashes from pieces of data information to identify presence of reference data in target data are researched in several projects [12–15]. This is the same technique widely used in digital forensics to identify whole files against a hash-set with a number of hashes to identify a group of files.

In law enforcement, the police widely use pre-defined hash sets containing known pictures and videos showing sexual abuse of children.

In digital forensics large sets of hashes are also used in data-reduction to eliminate files with known hashes from operating system and software installations.

To identify whole files using hash comparison is easier to understand as a valid method as the identified object is possible to verify by visual inspection.

Using a hash set from known sexual abuse pictures could be compared with hashes from pictures at the suspect's machine. When the hash comparison engine discovers hits, they are marked as identified child abuse in some way by placing these files in their own category of bookmarks etc. After such an automatic identification, procedures may require the operator to manually inspect a selection of the findings visually.

The use of block-hashing we could have hits on pieces from known material with limited abilities to visually approve the findings. This is the main challenge using block hashes as an identification method.

To approve the findings, we need to implement a set of rules to approve the hits. These rules must both have requirement to number of coinciding blocks and other factors that ensure the findings uniqueness.

By number of coinciding blocks, we mean requirement of certain number of equal blocks in the identification. We could use number of blocks or percentage of coinciding blocks.

By other factors to ensure uniqueness, we involves techniques and/or algorithms that approve each single block. Such proof could be the findings in sequential chains of hits, block-size and entropy.

## 8.1 Optimal hash algorithm to identify coinciding blocks

In Section 2.2.1 several hash algorithms are evaluated, both MD5, SHA1, SHA256 and SHA512. The MD5 algorithm several years ago proven not to be collision resistant and SHA-1 is no longer on the list of approved algorithms from US Department [10]. SHA256 is an algorithm approved to be collision resistant. An even stronger algorithm, SHA512 is also collision resistant. While MD5 that still are widely used generates a hash value of 32 bytes. SHA256 give a 64 byte hash string while SHA512 gives a string of 128 bytes. We could have choose the SHA512 to increase the strength, but there is no reason to use an algorithm stronger than necessary. By block hashing a 8 TB drive with 512 byte blocks, we will have about 16 billion hashes stored in a database. Only the hashes in this example will occupy 2 TB. If we index the same table on the hash column, the actual storage requirement will be 4 TB only to store the hashes. By using SHA256, the storage requirement is the half, 2 TB. This is a factor we need to take into consideration, and SHA256 is a compromise between strength and storage requirement.

Using weaker hash algorithm than SHA256 is evaluated but decided not to be an option.

## 8.2 Optimal Block Size to qualify the method

Several test performed in this project was done using different block sizes from 512 bytes to 16 KiB. By measuring entropy on each block generated, we have observed that the average entropy increases the bigger block we use. In Chapter 7.5 we have measured the average entropy on JPG pictures using block sizes from 512 to 8,192 bytes.

In Chapter 7.11 we have done similar test on an AES encrypted container using block sizes from 512 to 16,384 bytes.

In Chapter 7.12 the average entropy on ordinary text files is measured using block sizes from 512 to 16384 bytes. The dataset with block hashes from ASCII files is one billion records. Figure 7.26 show the distribution of average entropy for ASCII text.

Common to all of the three results measuring the average entropy using different block sizes is that the average entropy increases the larger block size we use.

To have sustainable findings between target data and the reference data, we need to base the major part of the findings on blocks with high entropy.

Collision detection is also performed on different data sets. In Chapter 7.12, the collisions on block hashes from ordinary ASCII text files are performed. The table shows decreased number of collisions as the block size increases. On the one billion record dataset, we had 25,000 collisions on block size 512 bytes and only 290 collisions using block size 16,384. We also observed that the maximum entropy changes significantly by using different block sizes. On block sizes 512 to 2,048 we had an maximum entropy of about 0.75. On block size between 4,096 nd 8,192 this drops to 0.15 and only blocks with entropy of 0 has collisions using block size 16,384 as block-size. This is shown in Table 7.13 and Figure 7.27.

Looking into Figures 7.3,7.25 and 7.26, we could easily think that using large block sizes is best as this will give less collisions, less number of blocks to search in and give the higher entropy.

To decide the optimal block size, we also need to look into the architecture of the storage to perform search on. On disk level, the default sector size has been 512 bytes for decades but now the default size is 4,096 bytes. Some devices have been observed with a sector size of either 8,192 and 16,384 bytes (Apple iPhone, iPod and iPad).

Modern file systems like NTFS, Ext4 and HFS+ use the term blocks or clusters to describe each allocation unit. Such a block is a collection of one or more sectors and forms the smallest storage unit for the volume. Among these commonly used file systems, the most used block/cluster size is 4 KiB. On more modern file systems like Microsoft ReFS (Resilliant File System) and Oracle ZFS, blocks sizes of 128 KiB are common.

On older file systems such as FAT16 and FAT32, clusters such a 16 and 32 KiB are normal and on the Ext 2 and 3 file systems, small block sizes such as 1 KiB are observed frequently.

When it comes to block hashing and the decision of optimal block size, we have to use a size adopted to the target for the search. If the search is based on a whole disk, we need to adapt the sector size used in that particular media. If the search is targeting a volume, we need to adapt the block size to the volume block/cluster size. The reason for such an adaptation is the the difference in minimum allocation units used on disks versus volumes. The minimum allocation unit could be equal, but this need to be confirmed.

A pit-fall in selecting a block size that is larger than the block/cluster size of the volume when processing a volume, or selecting a block size larger than the sector size when processing a whole drive, is the risk of making a block hash from an area occupied by more than one single existing or previously existing file. In this situation we have a file block contamination.

On the other hand, using small block hashing (512 bytes) will never have occasions of block contamination, but the side effect is the huge amount of blocks that need to be processed.

## 8.3 Entropy to qualify the method

Entropy as qualification to approve the block hashing method is partly discussed in previous section. All tests performed using different type of data and different block sizes are unambiguous. As we increase the block size, the entropy increases. This is discussed widely in Chapter 7.

Processing encrypted data, it is not uncommon to have entropy very near 1.00 and lowest entropy in this type of data is observed in Section 7.4.1 to be near 0.92 for 512 byte blocks, and 0.998 in 16 KiB blocks. Related to other types of data, this could be defined as high entropy.

In plain ASCII text by using the dataset in Section 7.4.2 the lowest entropy is 0.00 found in block hashes of all sizes (from 512 byte blocks to 16 KiB blocks). In this dataset, 33,000 of the one billion records has an entropy of 0.00.

Between the two examples above that we could use as upper and lower boundaries of entropy, we have other data sources such as compressed data in jpg pictures, raw data pictures, videos and pdf documents.

In these type of files, we both have high number of blocks with both high and low entropy.

Tests performed in this project also shows that collision in blocks of same size with very low entropy ( less than 0.5 ) is more likely to occur. In Table 7.3, a high number of collisions is detected in the Dataset-2 with block size of 512 bytes. The average entropy in the 162,292 collisions is 0.428.

In Dataset-2, nearly 12 million collisions are detected from a set of 50 million records with block size of 512 bytes. This is shown in Table 7.7. The average entropy on these collisions is 0.82. Nearly 1 million of these has an entropy less or equal to 0.5 and 750,000 records have entropy of more than 0.9.

In the 50 million records, nearly 9 million were blocks with entropy of 0.0 and all of them was blocks filles with only 0x00 values.

An important question is to interpret the influence the entropy have in this method. One option is to use entropy to implement what blocks to perform search in. Another approach is to use entropy to exclude blocks.

Using entropy to perform data reduction could save a lot of computing as the number of records are reduced. One option is to remove the records with zero entropy, but this could lead to unwanted results. Even blocks with zero entropy are still data. Such blocks of data are often found in the reference data we like to search for in the target and this could result in gaps in a sequence of common blocks.

If the method relies on having large numbers of common blocks sequentially, this kind of data reduction could interfere with that requirement.

In Section 2.4 the X-Ways Forensics and Guidance Encase tools claim exclusion of blocks with repeating pattern. This is not further described. This could be blocks with low entropy set at a certain unknown level, it could be only blocks with only equal bytes or by using one or more other criteria.

## 8.4 Continuous blocks to qualify the method

In a modern file system, the driver in the operating system are usually designed with algorithms that to some extent will try to store files in an un-fragmentet state. The Apple HFS+ file system has such algorithms implemented in OS X and will on the fly make sure files $\leq$ 20 MiB are stored un-fragmented.

A file are by default stored un-fragmented. As the time goes and the disk/volume fills up, there will be less continuous areas of free clusters/blocks to store large files. It is inevitably that these files will be fragmented. There is no guarantee that fragments are stored in numbered order.

When we perform block-hash search in the target data, it is always desirable to have many common blocks and preferably as a sequence of blocks. This apply both for target and source (reference) data too.

An ideal example is to have a whole file in the reference data found in the target data in a sequential non-fragmented state. This could be the fact in some occasions, but then we name it file carving with file-hash verification afterwards.

The other extremity is a few spread single blocks from the reference data found spread in the target. This is illustrated in the Figure 8.1

A more desired scenario is illustrated in the Figure 8.2. Here we have several block chains found in both data sets. The blocks are not only chains but also located in numeric order.

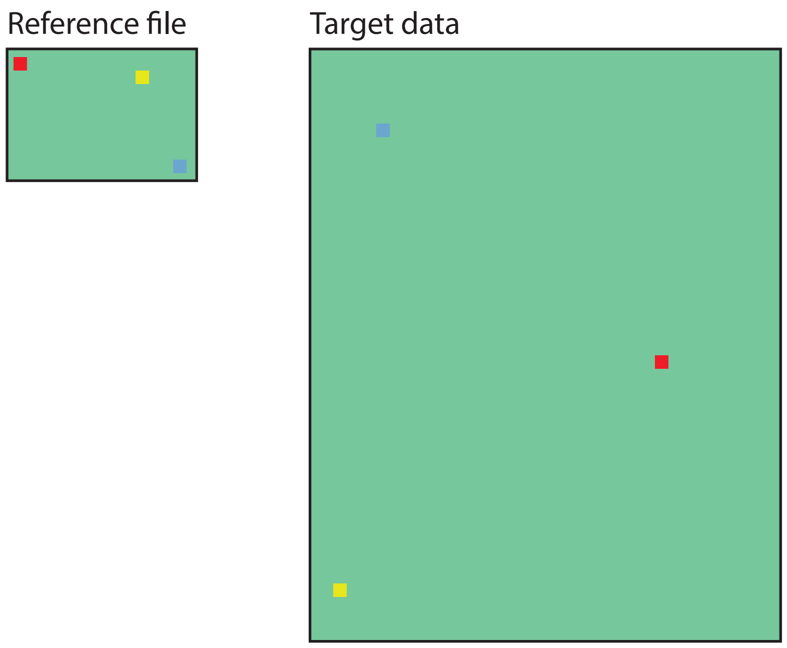


***Figure 8.1:*** *Example of few spread blocks non sequential and non continuous*

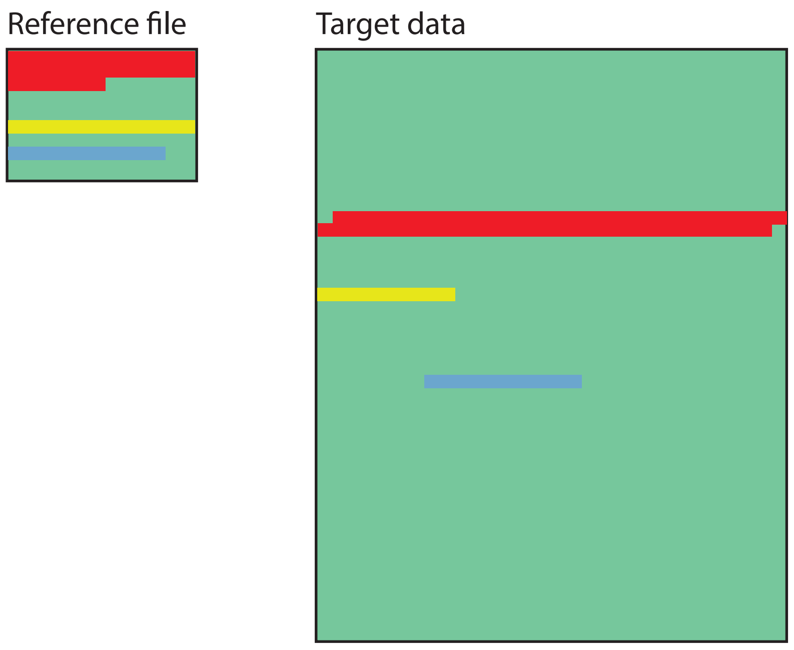


***Figure 8.2:*** *Example of large chunks of common blocks found sequential and ordered in both the reference and target data*

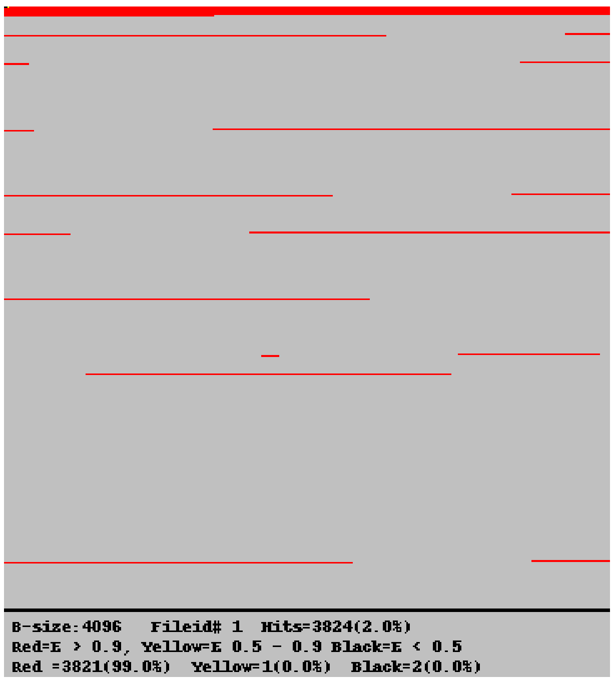
The experiments in Dataset-3, Chapter 7.3, we have several reference files with fragments located in unallocated clusters in a volume from a sharp case. An example is the reference file number 1 from Table 7.6 where $\approx 2\%$ of the blocks from a 537 MiB file is located in the unallocated clusters. The block-size used is 4,096 bytes. Figure 7.5 and 7.20 shows respectively the blocks in the reference file and the presumable same blocks in unallocated blocks of the volume.

Both Figures 8.3 and 8.4 are copies from Figures 7.5 and 7.20. We notice that all common blocks have an entropy of more than 0.9 and there is 10 continuous chains from the reference data we have hit on in the unallocated area. Totally, there is 3,824 common blocks. Most of these numbers are embedded in the figure.

On the next Figure 8.4 we have tree major areas in the unallocated clusters where we have located the common blocks found in the reference data. These tree areas are not continuous but reside in the same "neighbourhood".

In another example we use the reference file number 13, Figure 7.12 which is a map of the 4,096 byte blocks located in unallocated clusters shown in Figure 7.22.

These two figures is also shown here as Figure 8.5 and 8.6.



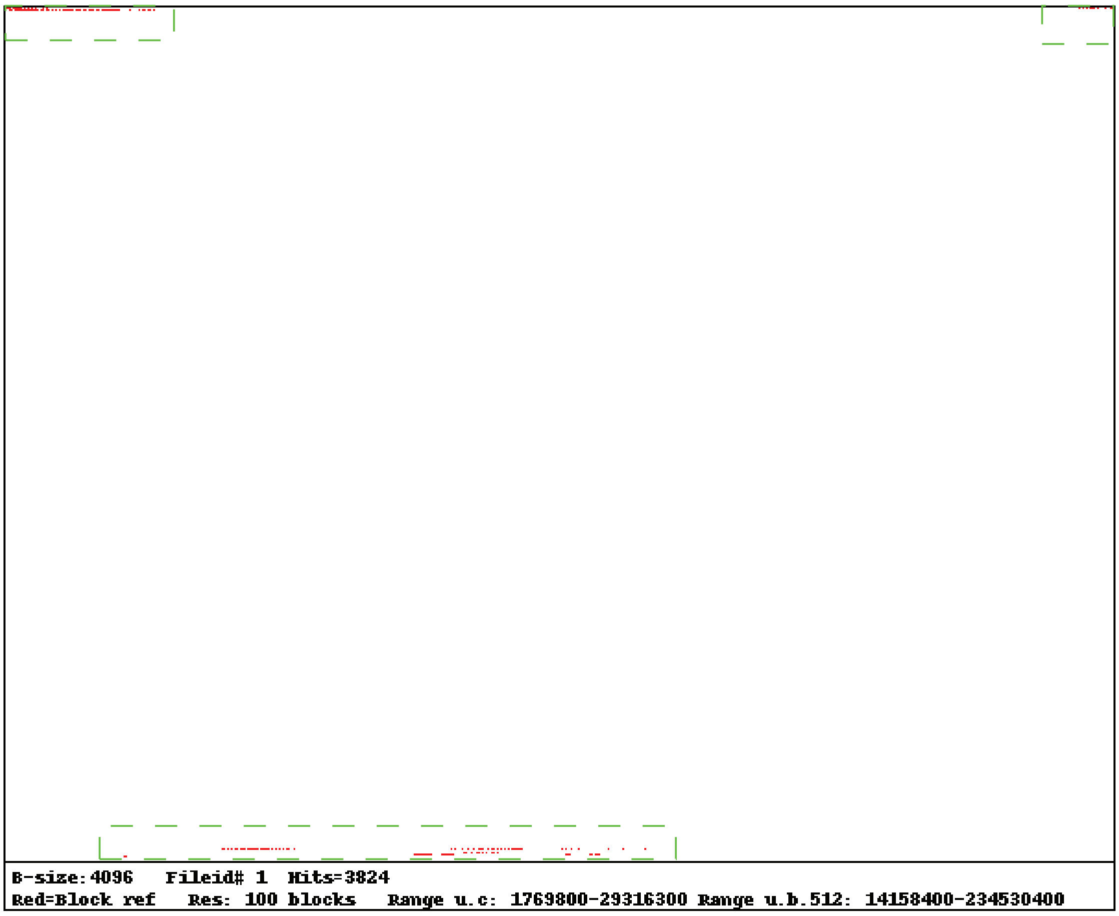
```
B-size:4096   Fileid# 1  Hits=3824(2.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =3821(99.0%)  Yellow=1(0.0%)  Black=2(0.0%)
```

***Figure 8.3:*** *block map of reference object 1, block size 4096*

Continuous blocks involves several blocks in a chain, and this could involve blocks with any level of entropy. In Chapter 2 we involved data reduction as one method to reduce the number of records in the data set. One method to reduce the amount of records is to filter out all records with low entropy. Earlier we discuss this in Section 8.3.

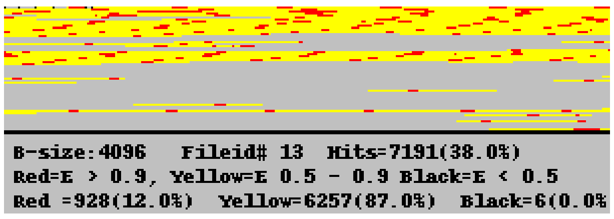
By using data reduction on the data set, we could end up removing blocks with low entropy to reduce the processing time but could result in having gaps in potential continuous block hits. We have also stated that there exists a lot of blocks with zero entropy in typical blocks with entirely repeating in the whole block (typically 0x00, 0xff etc.). Such blocks could also be part of a chain of continuous blocks. It is important to have this in mind in the seek of an efficient search to reduce the processing time.

One of the major question in this project is to determine number of continuous blocks or size of the continuous area these blocks represent.



```
B-size:4096   Fileid# 1  Hits=3824
Red=Block ref   Res: 100 blocks   Range u.c: 1769800-29316300 Range u.b.512: 14158400-234530400
```

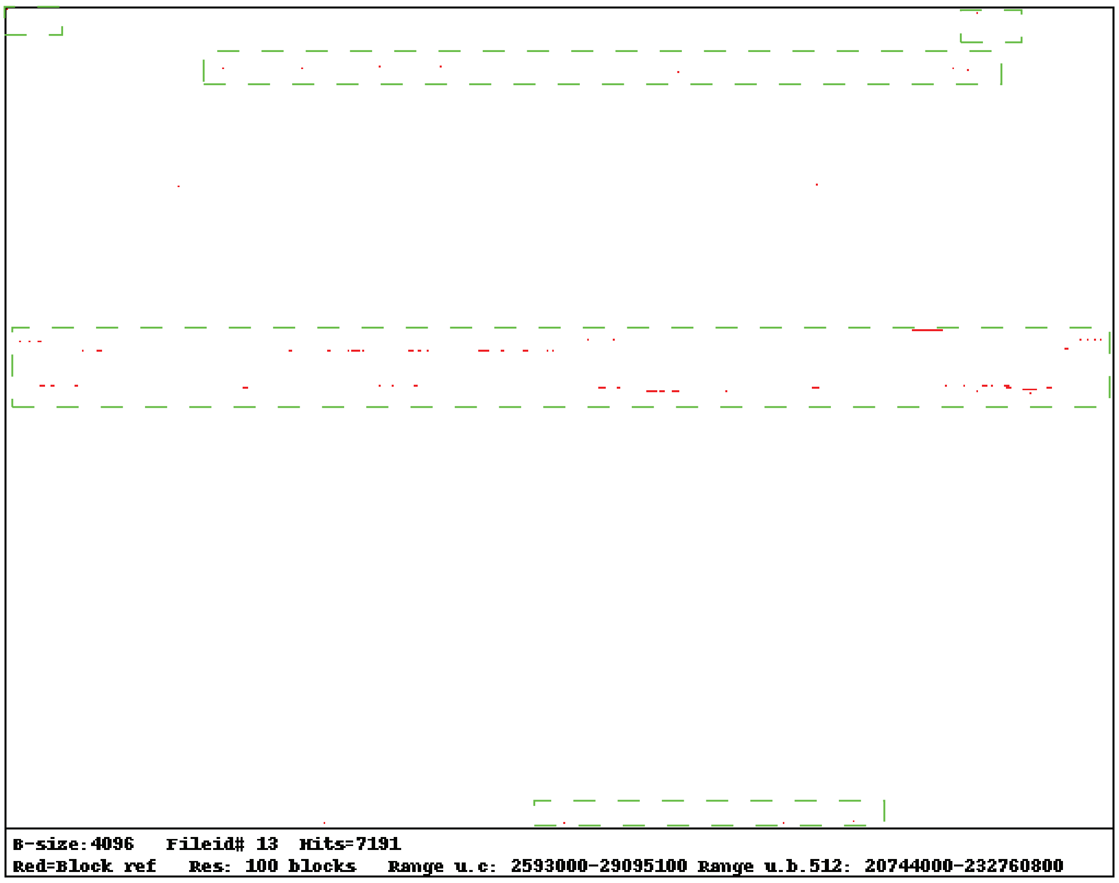
***Figure 8.4:*** *block map of placement in unallocated areas object 1, block size 4096*



```
B-size:4096   Fileid# 13  Hits=7191(38.0%)
Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5
Red =928(12.0%)  Yellow=6257(87.0%)  Black=6(0.0%
```

***Figure 8.5:*** *block map of reference object 13, block size 4096*

To determine the continuous value, we have to involve the factors block-size and number of blocks. As a simple example we could say that 1,000 continuous blocks of 512 byte block-size are as good as 125 blocks each of 4,096 bytes. The size of the continuous areas are the same, 500 KiB. Which of these two we use does not influence the result and is not important. A more important approach is to determine a minimum acceptable size of continuous areas. One method is to set the bias at number of bytes with coherent data. Another is to use percentage of the coherence between target and the reference data.



```
B-size:4096   Fileid# 13  Hits=7191
Red=Block ref   Res: 100 blocks   Range u.c: 2593000-29095100 Range u.b.512: 20744000-232760800
```

***Figure 8.6:*** *block map of placement in unallocated areas object 13, block size 4096*

The downside of using number of bytes is to relate it to reference data that could vary from a few KiB to several GiB or more and we could end up with setting a static value greater than the reference file. Using percentage of blocks related to blocks in the reference data could be more statistical correct, at least compared to using just number of blocks or bytes.

The challenge in using percentage, is to find a bias which is sufficient enough to work as an admissible proof in court. In Norwegian court of law, the use of predominantly evidence is defined as proof of more probable than not, often more than 50% certain.

To use the figure 50% directly against number of block hits, would not be correct. A more correct approach would probably be that a certain amount of coherence will make the evidence more than 50% probable. That number of blocks could be between 1-50%.

The amount of coherence should not be used as a stand-alone measurement to approve or dis-approve the findings. The most obvious approach is to combine this factor with factors like entropy and location in the target data.

Location in the target data is not mentioned specific earlier in the project, but is demonstrated using block maps from unallocated area in the Dataset-3. Earlier, we have mentioned that the goal of modern file systems is to store data not fragmented if possible. Very often we found data stored sequentially even on large files, at least when there is sufficient storage left. A file system like Ext4 will by default place all data per file in the same block-group if possible. Even if the file is fragmented, this policy applies.

## 8.5 False positives/hash collisions

In the evaluation part we have used several datasets with different behaviour and purpose. Dataset-1 to detect collision in a large database table with up to 15 billion records, the Dataset-2 to provoke collision to some extent and the dataset-3 to use a real case as a proof of concept of the technique and method. Some collisions are detected and expected due to the nature of the different data sets.

In Dataset-1 there was relatively low number of hash-collisions, $4.6 \times 10^{-5}$ in the table with 512 byte blocks and $3.2 \times 10^{-2}$ in the 4,096 byte block hash table.

In the Dataset-2 with more than 40,000 JPG files taken from a static background and give pictures visually equal. We detected only 0.14% collisions in the 512 byte block table and 0.07% in the 1,024 byte blocks. Larger blocks have no collisions detected. When converted the same pictures to BMP-2 and exclude the 56 byte header, we detected no collision using any block-size. The BMP-2 data is just a bunch of RGB color numbers. No collision detected in these blocks was a bit surprising.

The main hypothesis is based on low grade of collision in data-blocks at any block-size. We have set 1.0% and below as low hit-rate.

In the Dataset-3 we have 112 GiB of data from unallocated area of an NTFS system volume (Windows 7) from the suspect's machine. In this dataset we detected a high number of collisions, and was expected. The unallocated areas will normally have large areas filled with repeating patterns like hex 0x00 or other equal bytes. For the 4,096 byte blocks table in this data-set we found 823,877 blocks filled with zero value bytes. For the table with hashes from 512 byte blocks we detect a collision rate at 6.1%. This number of collisions is above the bias of 1.0% and undermines the main hypothesis.

In the unallocated areas we also find chains of blocks with coherence from the Dataset-1. We found traces of 64 of these videos , and several was found in more than one instance. About 600,000 blocks (4,096 byte blocks) was related to the video files in Dataset-1.

The unallocated area is a bit unuasual compared to the allocated area. It is usually not continuous and is very volatile. According to [34], modern operating system like Windows NT and above creates numerous files during runtime where Vogel's tests show that 80% of newly created files has an average lifetime of 4 seconds. Thousands of files, often very small with arbitrary content is created and deleted regularly, even in idle time.

Another matter about the unallocated area is the de-fragmentation processes on modern OS where the OS either automatically according to a schedule or manually is de-fragmenting files. This means files, usually larger files are partly re-allocated with a purpose of making all clusters sequential to increase the overall performance of the computer. During the de-fragmentation process, part of files are copied to new locations and the copied clusters still remain unchanged in unallocated. If the same file are de-fragmented more than once, there are possibilities of having same blocks repeated in unallocated area. If a file is re-allocated once, each block is moved from unallocated area to allocated and one from allocated to unallocated.

The Dataset-3 was not initially intended used as test source to detect collisions and probably not suitable to use as source for such testing, at least when this source is from a system disk of a modern operating system but also generally. Potentially, we could have occasions where a user erase the same file from two locations. If this had been a video file of 1.5 GiB, we will have 3 GiB of data changed from allocated area to unallocated. In this scenario on the same disk, this is about 2.5% of the unallocated area and would have ended up i our test statistics as block collisions.

Due to the nature of the unallocated area, he high number of collisions in this area are not in conflict with the main hypothesis.

## 8.6 Other factors with influence of the method

There are several other factors that influence the method. One of the important factors to discuss is the data reduction. This topic is to some extent mentioned in the Chapter 2 and 7.

By using block-hashing, we divide the data into equal pieces and create a database-record for each of them. Such a record by using SHA256 as algorithm, will have a record length in the database at around 100 bytes. By dividing the data into 512 byte blocks, we actually put $\frac{1}{5}$ of the whole data set into the database. To perform effective searches in the database, we need to index on the SHA256 value and the storage per record then increase to around 164 bytes per 512 byte block hash. This is nearly $\frac{1}{3}$ of the whole dataset. By using larger blocks like 4,096 bytes, we decrease the ratio from $\frac{1}{3}$ to $\frac{1}{25}$.

Even by using 4 KiB blocks we use severe amount of data storage and processing time. Data reduction could be an option to reduce this. We have suggested using some criteria to exclude certain blocks from the database. This is criteria like blocks with low entropy or blocks from the allocated area to exclude blocks from the unallocated area on the volume.

Technically, this process is straight forward, but have some side-effects not desired. By removing blocks using any criteria, will presumable came into conflict with one of the most important criteria in the method, the continuous blocks factor. Even by excluding typical zero-entropy blocks (typically blocks with repeating patterns like 0x00, 0xff etc), we potentially broke larger continuous chain of blocks into two or more minor chains. The challenge and side effects in using data-reduction is demonstrated in Figure 8.7.

The figure demonstrate what impact the data-reduction could have when we exclude blocks (records in the database) according to some criteria. In this example, the data-reduction removes blocks that initially could have been part of a chain of common blocks. After the reduction, the potentially "one single large chain of blocks" is reduced to tree separate chains. This could degrade the evidential value of the evidence.

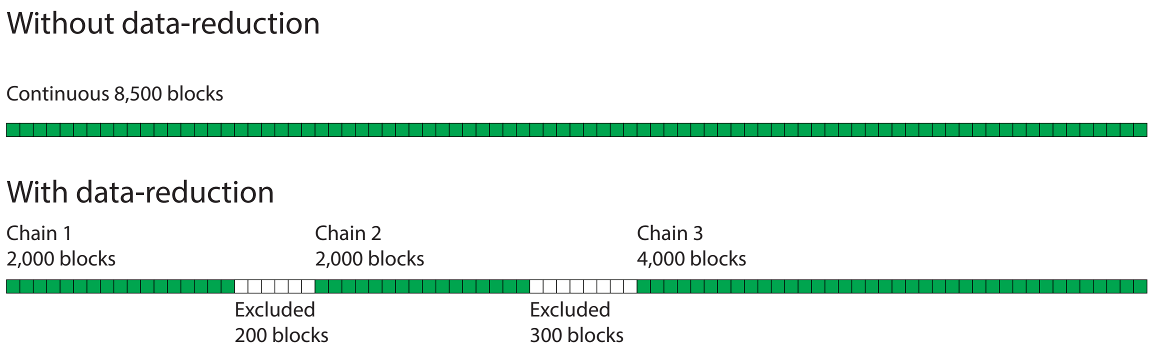


***Figure 8.7:*** *Data-reduction possible side-effects*

To overcome the problem with broken chains when doing data-reduction, we could set rules in the processing. Such a rule could evaluate several chains. By defining a reasonable gap-factor between chains, we could as an example conclude that two chains with a gap of less than a certain amount of blocks is one continuous chain of blocks. This is technically possible but is a speculative method of making "Things to look better" and is forensically not a very sound way of interpreting evidence.

## 8.7 The combination of block-size, entropy and continuous blocks

To approve the block hashing as a forensic method we have discussed important factors to assure the robustness by looking into quality and quantity factors as block-size, entropy, false positives and amount of block-coherence, four factors. Each of these factors play a vital role to evaluate evidential value the finding have.

In an ideal world we would have hits between reference data and target with many sequential continuous blocks using large blocks all with high amount of entropy with almost absence of block collisions in the target data. This is optimal but unfortunately not very realistic. A more realistic scenario could be (the list is numbered, not ranked):

1. High grade in 3 of 4 factors
2. A reasonable high score within all factors.

In the first option we could pick tree factors but then we have a challenge in ranking the importance of each factor. A suggested order of the factors after several test and the evaluations in this chapter should be as follows ( 1 is most important, 4 is least):

1. **Amount of block coherence**
   High number of common blocks found sequentially
2. **Block size**
   To use as large blocks as possible without exceed the cluster size in the file system if the search is limited to volume. If limited to a whole disk, do not exceed the sector size.
3. **Entropy**
   If blocks are found in small continuous chains (typical between 2 and 10), then high entropy could be important.
4. **False positives (block collisions)**
   decided as the least important factor as collision after several searches is not detected in high grade and have less importance if the previous factors are in use.

The tree first factors in the ordered list is both individual and dependable, particularly 1 and 2. We could have occasions where we have a reasonable high number of large blocks continuously that have the same number of bytes on a higher number of smaller blocks in a chain. As an example, 1,000 blocks of 4.096 byte blocks have the same amount of common bytes as 8,000 continuous blocks each of 512 bytes. The common area in both examples are 4 MiB. However, the average entropy for the 4,096 byte block example will be higher than the average for 512 byte blocks. This is illustrated in Figure 8.8.

The two examples should normally have the same evidential value.

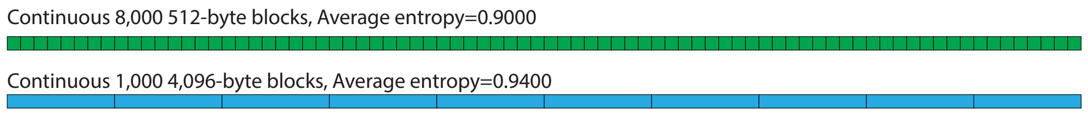

Continuous 8,000 512-byte blocks, Average entropy=0.9000
Continuous 1,000 4,096-byte blocks, Average entropy=0.9400

***Figure 8.8:*** *Example of continuous blocks 512 vs 4,096 byte blocks*

Another example from the Dataset-3, Chapter 7, Figure 7.13 where 7.0% of the blocks in the reference file 20 i located in the unallocated area of the target data. In this example, the finding are spread trough the whole reference file with just small chunks from 2 to 10 common blocks in chains. Most of the blocks has medium entropy except an areas in the middle of the picture with 1,344 blocks more or less in chains from 10 to more than 50 blocks.

In this example, the continuous chains is not large but there is a large number of them. If we compare this with the location in the unallocated area, we found the common blocks in a few limited areas ( 5 major areas). This is shown in Figure 7.18.

In our third example, we use the Figure 7.6 where 9 % (5,406 blocks) of the 4,096 byte blocks from file 9 from the reference data is located in the unallocated area. Here we have large continuous areas and all common blocks are of high amount of entropy. The largest chain of blocks here are 2,247 blocks ( 9.2 MB) with entropy > 0.9.

## 8.8 Criteria for documentation

In the Section 2.4 about known commercial tools supporting block-hashing we have an example from X-Ways Forensics. To some extent, the findings is documented with cluster of volume the hit is from, what hash-set the hit is from and the size of the continuous blocks given in KiB. To verify that the hit actually is from a certain file or other location, the operator needs to compare the target and reference data manually. On text documents this give no major challenges, but when it comes to data with more random looking content, this is more demanding. The hit list does not contain information about the hash of the target and/or reference data. There is no reference to block number in the reference file.

In computer forensics the ability to verify evidence is crucial to ensure the validity of the traces. We often talk about best practise, and that is procedures adapted trough a period of time. When it comes to block-hashing, we have found no references to best practise.

The documentation of digital evidence is often a challenge with no golden standard. In the teaching at NPUC we often tell the students to document enough but not to much and not to little.

To ensure the findings is possible to verify, we need both reference to each block in the target and the reference data. The reference should at least contain information about:

- Block-size used
- Complete reference to the block in the reference data (reference object type and internal block number)
- Complete reference to the block in the target data (type of object and internal block number)
- Characteristics about the block (hash value and entropy)

An example of database with references described above is found in the Table 4.1 and 4.2 in Section 4.1. Similar type of references is used in the Dataset-3 with target data described as block number in unallocated area from a volume.

## 8.9 Verification of findings

As stated in previous section, the importance of having a system that generates evidence, also need to have information make verification possible. By identifying common blocks with exact location both in reference and target data is crucial to make verification possible.

We have already done several test showing that the technique works and the same is proven in earlier works. The whole process in block hashing is a kind of verification by comparing hash values, but this is on block level. To have an over-all verification, we need to compare the original reference file with some common data usually found in unallocated area.

One method to verify is injection-verification where the common blocks found in unallocated is injected to a copy of the original reference file. Since our method involves information about location in target and reference file with the SH256 as the link, it is possible to put the common blocks from unallocated area into the copy of the reference file. After this process, we can compare the file hash of the injected copy with the original file.

The example SQL command is found in A.16 and in Table 8.1 is an example listing.

Figure 8.9 illustrates and example where we have the original reference file # 1 up-left which is the complete file with a known MD5, H1. Up-right is a copy of the same reference file which is injected with the common blocks between the reference and target data. After the injection-verification, the injected file will have a MD5, H2.

If H1 = H2 after the injection, we can be sure that the common data found in unallocated area is identical to the same data in same position in the reference file are equal.

| Referene file # | Reference Block # | SHA256 | Unallocated Block # |
|---|---|---|---|
| 1 | 1933 | d5fafe4e897abfd57921ee2af7bb30223ec3147adbb1faf7607052338b1e49fc | 29104408 |
| 1 | 1934 | 34ec7298df143d1937e05ab5c853726a5b3bc42e9510fad58215fbe2bec92898 | 29104409 |
| 1 | 1935 | 6556eb43b1106f84315dbbf9d362fb827d538e4f324245888315c7c10ed9e492 | 29104410 |
| 1 | 1936 | c14b4dd3d72852ec51d0753db83d6619540b34c6d0f302242acc264708803cbb | 29104411 |
| 1 | 1937 | de0b3b1b3b9907b121a5cee490617c80fe8affc0a4a45837e16d8d0649e53c83 | 29104412 |
| 1 | 1938 | 8c2ef1bdedc7664dd7a9ab5c9b6ebc23cdc215cfdda5043cd768c33eea2a4ea6 | 29104413 |
| 1 | 1939 | f719adf8ffb19a8ced46328ea9af200b636718b3db4fed8446f7d3eb1bb8e816 | 29104414 |
| 1 | 1940 | c6cfdae3ffd465370ee95074b41d9aaab77c090a6fbbd0b9564cec7fddea9f92 | 29104415 |
| 1 | 6144 | e98920b04fbf311ae536df262cd937ebbafcf62741839e64e01ada87623ff112 | 1835788 |
| 1 | 6145 | 0fa158bf097ab51e4cd624e3204ba9c1576220153368eea7a41944502fe307a8 | 1835789 |
| 1 | 6146 | 5b099fcbddb091fa1e5a28a8f4242aba314c889645da1269bf11883439994606 | 1835790 |
| 1 | 6147 | f10f9cbb7058756cec4bb2869b5a2469bf8dc33501a79fd1e6818dfb6066eef5 | 1835791 |
| 1 | 6148 | 55b094f2c1ebe6cbbe556929bf3404272982a9cb68b20d8e6894c64cccb7718c | 1835792 |
| 1 | 6149 | f8da56674acf5abc41777f00a95dd1b70a9a39cf0e46b6c24b563cb7da8f9603 | 1835793 |
| 1 | 6150 | d93b4d0ed04c7859a1cfe21cf7d497322ee8652c993eab31faff2611fd2c1929 | 1835794 |

***Table 8.1:*** *Detection of collisions in video file blocks in Dataset-1, block-size 4,096*

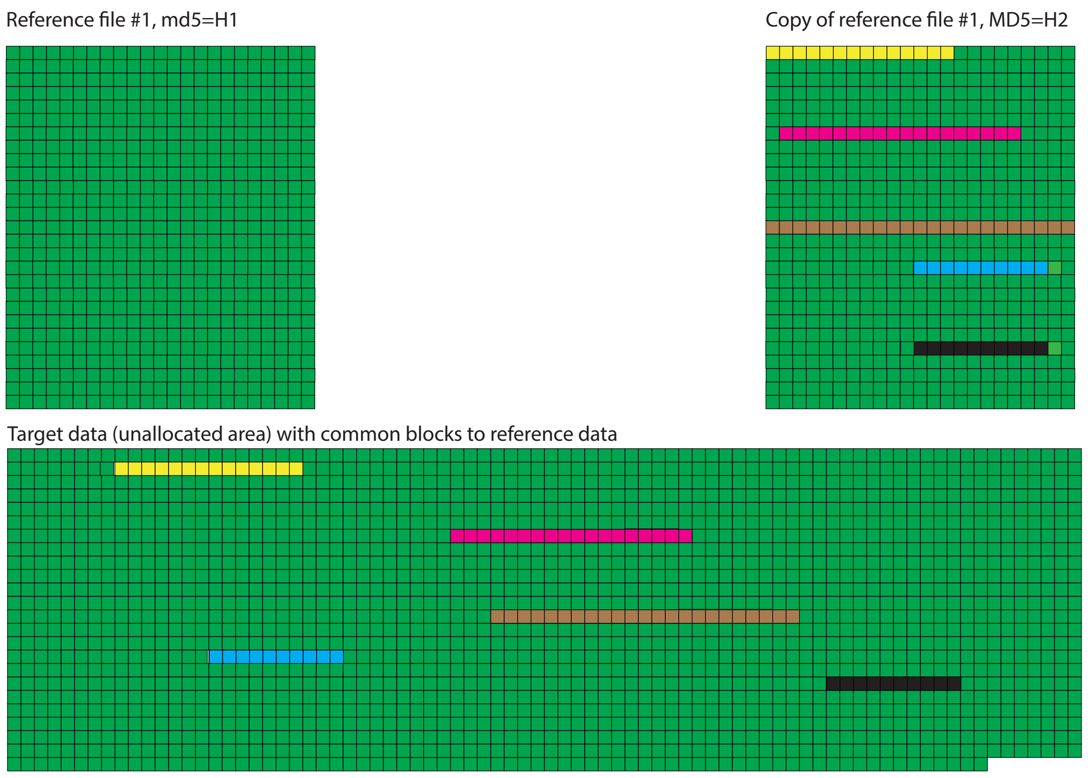

Reference file #1, md5=H1
Copy of reference file #1, MD5=H2
Target data (unallocated area) with common blocks to reference data

***Figure 8.9:*** *Reconstruct with verification common block from reference and target data*

Table 8.1 is the result of a SQL Union search between reference and target data blocks of same size The table shows 15 hits and is an example of sufficient documentation of finding. We have the SHA256 hash value in column 3 that connects the data from target and reference data. Next we have exact reference to the reference file in column 1 and 2 which is the exact reference file with block number. In column 4 we have the blocks number in the unallocated area. These values are sufficient to use as verifiable documentation of the findings.

## 8.10 Other factors influencing the method

One of the goals on doing this project was to use only open source tools. As database engine, MySQL Community version were used. One major drawback on non commercial database-engines is the lack of ability to perform operations using multi-threading searches in database tables.

In the largest dataset, the Dataset-1 in Chapter 6.1.2.1 the dataset contains respectively around 2 and 16 billion records per table (table with 4,096 and 512 byte blocks).

One of the tests, detection SHA256 block-hash collisions, the search in the 2 billion record table was aborted after 87 days. The search would probably have taken around 100 days. Statistically the same search in the 16 billion record table would have taken eight times more time to perform, more than 2 years.

After the search was aborted, the same database was ported to a Microsoft SQL 2016 server and the same search was done in a fragment of time. The collision detection took respectively 44 minutes and 11 hours on the same hardware as the MySQL engine.

On Dataset-2 we do collision detection on different block sizes and on the largest dataset, the database table from 512 byte blocks from the pictures converted to BMP-2 with about 236 million records, the MySQL finish this in 13,212 minutes while MSSQL 2016 do exactly the same search within 33 minutes. That is a speed of $\frac{1}{400}$ and is an important element when doing such investigation to avoid using more processing time than needed.

To use block hashing as a forensic method, one of the key elements in performing such analysis in a reasonable time is the hardware and software, specially the database engine. It seem undoubtedly that the process depend on a fast and reliable database engine like Microsoft SQL server, Oracle Server, IBM DB, Sybase or other enterprise database solutions.

Running the method on small storage from a home computer of 100 GiB could be solved using community versions of different databases but as the storage increases, even on a regular home computer, default storage now are often more than 1 TiB. This is also according to Moores Law by Gordon E. Moore [1].

The datasets was created using Python scripts and the processing of the target and reference data could have performed faster with using compiled C sources to executables.

[1]http://www.mooreslaw.org

# 9 Conclusion

In this chapter we will try to answer the reasearch questions stated in Chapter 3 and discuss if we have archived the goal of this project.

The answer is based on the experimental work in part III and discussion of the results in Chapter 8.

There is one main research question and five sub-questions, Chapter 3.

Several test was performed using the tree datasets. The following major key questions is evaluated.

## 9.1 Is block hashing a recommended, sustainable method to identify presence of the reference data to use as admissible evidence in court

### 9.1.1 Define criteria to ensure blocks in reference and target data are the same

As stated in Chapter 2, The MD5 algorithm is not on the list of the US Department list of secure hashes [10]. Today, the more secure hashes are the SHA-1 [18], SHA-2 [32] and SHA-3 [20] algorithms. SHA-1 [18] is not longer on the same list of secure hashes. The SHA256 algorithm is decided as the best alternative and no collisions are detected on this yet. There are other more secure hashes like SHA512, but there is no need to use more strength than necessary. The SHA512 hash is a string of 128 bytes while SHA256 is a 64 byte string and therefore need half of the storage capacity.

Further description and discussion on different hash algorithms are in Section 2.2.1 and 8.1.

After intensive testing in Chapter 7 and discussion in Chapter 2 the probability of false positives are relatively low. In the largest set of data, dataset 1 in Section 7.1, Table 7.2 a collision frequency of less than $3 \times 10^{-5}$ is detected both on block size 512 and 4,095 bytes.

### 9.1.2 Setting bias for amount of mutual data between reference and target data

To determine necessary amount of coherence between the reference and target data, we have several options. One option is to set a minimum number of blocks with equal hashes. Another approach could be a percentage of the reference data. A third option could be a combination of these to in addition to the behaviour or the mutual blocks.

The common blocks from the two datasets could either be a spread of single blocks or chains of blocks with an arbitrary length. One of the goals of modern file systems is to keep files un-fragmented if possible. The HFS+ file system policy claims files smaller than 20 MiB are kept unfragmented on the fly. By taking this under circumstance, we can assume that a major part of the files on modern file systems are stored in one contiguous chain of blocks/clusters. If this approach is not possible, the file systems will try to make as few fragments as possible. There will be exceptions from this. Typically large files from databases and virtual disks are meant to grow over a period of time. These files are often seen heavy fragmented but is more an exception from the general rule.

When files are erased, the used blocks in the file system are available for new content and as the time goes, these areas are filled up with new data. Still, we can expect to find contiguous file system blocks/clusters from the erased file. To find only single blocks spread all over the unallocated area is unlikely, especially the large files like digital photos and video.

In a normal situation, we could expect to find larger areas of contiguous common blocks in addition to some occasions of single blocks spread in the unallocated area.

In dataset 1 ( the msc_case database) we have an example from reference file number 20 from Table 7.6 in Figure 7.13. This is from a video file of 52 MB where 7% of the content is located in one large contiguous chain of blocks in addition to more than 1,000 small fragments between one and five blocks. Common to most of the small fragments are the amount of entropy. About 15% of the hits have low entropy and nearly 60% have medium entropy ( between 0.5 to 0.9 ).

The file 20 have an area of about 2,000 contiguous 512 byte blocks. This is nearly 2% of the total number of blocks from this file. If we look at the block map for the same file but now with blocks size of 4,096 bytes. This is Figure 7.8. If we compare this figure with the previous one covering 512 byte blocks, all the small fragments of one to five blocks are omitted since each of these small chains only represents from 512 to 2,560 bytes. Still we have the large contiguous area of 4,096 byte blocks. There is nearly 2,000 common blocks in an almost non-broken chain. These 2,000 blocks are nearly 2% of the blocks for this reference file.

If we look into file number 1, 9, 13, 20 and 24 from the Table 7.6, we have block maps for Figures 7.5, 7.6, 7.7, 7.8 and 7.9. These figures shows the 4,096 byte blocks from the reference data located in unallocated clusters. All these files have large contiguous areas of common blocks between 1 and 38% of the blocks in the reference files. Figure 7.9 is from file number 24, and there are two block chains covering the mutual blocks. Even if this is only 1.5% of the reference data, this is 564 common blocks, each of 4,096 bytes. This is 2.3 MB out of 152 MB.

For the file 24 we also have a block map of where these common blocks are found in unallocated areas. If we look into Figure 7.24, we notice two areas in unallocated areas where these common blocks remain.

The file 24 using 4,096 block size, is a good measure to define a bias of how many blocks we should set as a minimum to approve that the found blocks in unallocated areas most certain have been part of the same file as in the reference data.

To further test this hypothesis, we make a copy of file 24 and wiped all common blocks in that file. That was block 28,500 to 28,585 and 32,290 to 32,767. After this operation we injected the common blocks from unallocated ares into the copy of file 24. After this we compared the MD5 hashes of the original file 24 and the injected copy. Both have the same hash "fb57452f4a41bd9e7906dac819a63920". This method to verify, we have named "Injection Verification" and is further described in Section 8.9.

By selection two of the files with least amount of common blocks in a chain, it seems like 1% common blocks in a contiguous chain is sufficient to approve the partly presence of a reference file in unallocated areas.

In the examples with file 24, we have both stated the common blocks are in chains both in the reference file and in unallocated areas. The entropy is above 0.9 for most of the blocks and we have also performed an block injection described two paragraphs above.

### 9.1.3 Optimal block size to use

Several block size evaluations are performed in Chapter 7. Previous work have tested block-hashing using different block-sizes, mainly 512 and 4,096 bytes. In this project we have conducted tests on many more blocks sizes, both 512, 1,024, 2,048, 4,096, 8,192 and 16,384. We have compared the average entropy on different block sizes, detected block collisions using different block sizes and used a real case to test the method in full scale.

We have observed several impacts using different block sizes. First of all, the tests are consistent about entropy and probability of block collisions when increasing the block size. All conducted tests shows an decrease of block collisions when increasing the block size and this is significantly when increasing from 512 to 4,096 byte blocks. The collisions are severe only by increasing the blocks size from 512 to 1,024 or 2,048 bytes. All tests on entropy are significant. The entropy increases as the block size increase, and blocks with high entropy normally are more resistant to collisions.

Another approach is to define the optimal block size. It is undoubtedly that the larger blocks we use the more consistent the hits are. However, there is a limitation. It is not recommended to use larger blocks than the smallest allocation block/cluster size the target uses. If we are performing searches with a volume as target, we have to avoid larger blocks than the block/cluster-size used in the volume. This is to avoid block contamination where data from two or more previous files could be part of a single block. By example using 16,384 as block size to search for, we could end of having data from up to four different files in one one single block when processing a volume with 4,096 byte clusters. The same is an approach when a disk is the target. Still many disks use 512 byte sector size and performing search using block size greater than 512 could end up with fragments from many files in the same block. Before we start processing a target object, we need to analyse the structures of the target data.

Today on several of the modern file systems like NTFS and HFS+, the most common cluster/block size are 4,096 bytes on these. On even more modern file systems like ReFS, ZFS, btrFS and APFS (APFS is the new file system used by OSX), we can expect to find cluster/block sizes of either 32, 64 or 128 KiB. In these file systems, block-hashing can be conducted using even larger blocks than tested here.

The selection of block-size have a severe impact of processing the target data, both in number of records to handle in a database but also in storage requirements. When block-hashing terabytes of data, the decision of which block-size to use may have a severe impact of search time and storage requirement.

### 9.1.4 Other factors to approve or disapprove the method as robust enough

An important approach is to evaluate practical issues. Compared to regular file hashing and comparing, block-hashing is far more resource demanding and there are larger hardware and software challenges. Regular file hashing and comparing, usually rely on a predefined hash set from known files and we usually don't need do generate these. In block hashing, it is not feasible to use the same method even if it's possible. We will have to create the block hashes from the reference data on each case. In addition, we need to create hash sets from the target data, usually the unallocated areas. In normal file hashing, we generate a hash per file and that is generally very few comparing to block hashes.

In block hashing the comparing of hashes involves multiple more hashes than in regular file hashing and the reference data also involves multiple more hashes than normal file hashing. In dataset 1 we have one database table with 15 billion 512 byte block-hashes from 8 TB of video files. In a normal case to compare that large number of blocks with another database table with an arbitrary number of block-hashes, require both a lot of processing power and software (usually a database engine) able to handle a cross query between those tables. We have already tested this and have experienced the importance of the database capabilities. Using a database engine not capable of utilize all resources on the physical machine, is not recommended and the search is not feasible.

Entropy is a topic described thorough in this prosject, bot in Section 2.2.2 and 8.3. Entropy can both be used to exclude blocks-hash records from the database. Entropy can also be used as a filter to what findings to include in the list of evidence. In Section 8.3 we have already stated that data reduction probably will interfere with the most important criteria in this method, "contiguous blocks". To use entropy to evaluate the evidential strength of mutual blocks are more realistic. We have already observed in several tests that the probability of block collisions are more likely on blocks with low entropy. To define an entropy value as high or low, we need to put the value in context of what type of data we search for. Are we searching for data objects typically compressed or encrypted, entropy below 0.9 must be considered as low. If the data we search for are plain ASCII text, we may consider defining high entropy as above 0.7.

In Section 9.1.2 we have concluded that chains of contiguous blocks in the findings are better than single blocks or small chunks of blocks. By adding entropy into the evaluation, we can strengthen the evidential value these findings represents, or opposite, reduce the evidential strength.

### 9.1.5 Verifiable

In Section 8.9 is discussed and the ability to verify evidence are crucial in all digital forensic work. By verifiable we both mean the ability the initial investigator that conduct the block-hashing, or a third party person has to verify the findings. To make such verification possible, we need to have sufficient documentation. This is further described in Section 8.8, and these recommendations should be a minimum. The finding have to be documented with the following information:

- Block-size used
- Complete reference to the block in the reference data (reference object type and internal block number)
- Complete reference to the block in the target data (type of object and internal block number)
- Characteristics about the block (hash value and entropy)

An example of documentation after a block hash search is in Table 8.1

### 9.1.6 Is it feasible to combine the above criteria to ensure the findings are admissible evidence ?

The different criteria are hash algorithm to use for block comparison, block-size, contiguous chains of blocks and entropy are factors in combination that would have the potentiality to ensure findings are of high evidential value and usable as admissible evidence in court of law. The selected hash algorithm, SHA256 is strong and approved to be collision resistant and is one of the accepted algorithms approved by US Government. By ensure the and optimal block-size is used and coincide blocks are mainly contiguous with more than a few blocks, it should be sufficient to use as admissible evidence. To additionally increase the strength of the evidence, we recommend using entropy as a factor to qualify the finding. The entropy must be adapted the kind of data we search for.

In addition, the documentation must be sufficient and as described earlier in this chapter and further in the Section 8.8. We also recommend that a reasonable number of finding are verified manually or by using software dedicated to such verification.

## 9.2 Investigative skills

Using block-hashing to identify previously erased content are most relevant when using the unallocated areas as target for the search. The method is not only about finding equal blocks between two datasets but involves many aspects and several elements must be evaluated. To make sure the finding are admissible as evidence in court, the investigator need extensive skills evaluate the findings and set criteria that make the evidential value strong enough. The investigator need deep knowledge about the architecture of disk, file systems, the unallocated area and different file types. The investigator must be capable of defining correct block-size to use and evaluate the relevance entropy have on different types of data.

The block-hashing method to locate previously erased data in a target data area based on blocks from reference data is illustrated in Figure 9.1

## 9.3 Further work

Research tries to answer the research questions by include a set of parameters to ensure finding after a block-hashing location of previous erased data. Using this method of searching for blocks of erased data involves a huge amount a storage and processing capacity in the analysing environment. During a research, it is not uncommon that the research yields more questions and therefore new research.

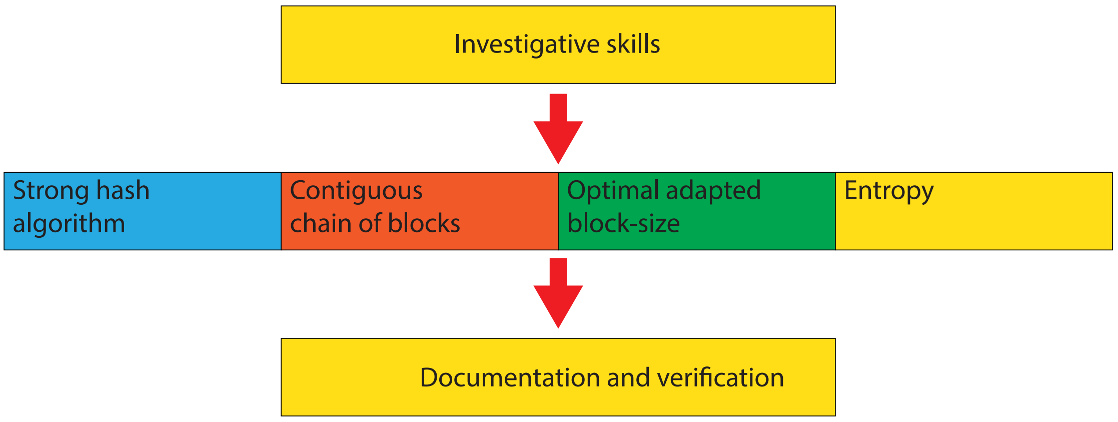

Investigative skills
Strong hash algorithm
Contiguous chain of blocks
Optimal adapted block-size
Entropy
Documentation and verification

***Figure 9.1:*** *The block-hashing method illustrated.*

Performing this kind of investigation involves a lot of processes to create the datasets, particularly to process, generate and fill the databases with block data. Next the actual search and finally the verification.

Today the exists a few tools performing this kind of analysis and these are referred to in Section 2.4. These tools are not according to the set of criteria in this project. Building a software framework with the block-hashing implemented to meet all criteria defined here would be a natural extension of this project.

In this project, we have only used one single real case. The method should be more intensely tested against real cases with an increased focus on verification of findings.

# Bibliography

[1] *Handbook of Information Security, Information Warfare, Social, Legal, and International Issues and Security Foundations.* John Wiley and Sons, 2006.

[2] *File System Forensic Analysis.* Addison-Wesley, 2010.

[3] Jason Beckett and Jill Slay. Digital forensics: Validation and verification in a dynamicwork environment. 2007. Last visited 2016-06-08 at https://www.computer.org/csdl/proceedings/hicss/2007/2755/00/27550266a.pdf.

[4] Brian Carrier. Open source digital forensics tools the legal argument 1. 2009. Last visited 2016-05-03 at http://www.digital-evidence.org/papers/opensrc_legal.pdf.

[5] C. G. Chakrabarti and I. Chakrabarty. Boltzmann entropy : Probability and information. ., 2007. Last visited 2016-04-21 at http://arxiv.org/pdf/0705.2850.pdf.

[6] Colinn Chisholm. Integrating forensic investigation methodology into ediscovery. *SANS, 2010.* Last visited 2016-04-21 at https://www.sans.org/reading-room/whitepapers/incident/integrating-forensic-investigation-methodology-ediscovery-33448.

[7] Sylvain Collange, Yoginder Dandass, Marc Daumas, and David Defour. Using graphics processors for parallelizing hash-based data carving. *IEEE Computer Society*, 2009. Last visited 2016-04-21 at https://hal.archives-ouvertes.fr/file/index/docid/350962/filename/ColDanDauDef09.pdf.

[8] Microsoft Corp. Default cluster size for ntfs, fat, and exfat. 2016. Last visited 2016-04-21 at https://support.microsoft.com/en-us/kb/140365.

[9] Seagate Corp. Transition to advanced format 4k sector hard drives. 2011-2012. Last visited 2016-04-21 at http://www.seagate.com/gb/en/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/.

[10] USA Department of Commerce. Fips pub 180-4 secure hash standard (shs). ., 2015. Last visited 2016-04-21 at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[11] Stefan Fleischmann. X-ways forensics/winhex manual. 2016. Last visited 2016-04-21 at http://www.x-ways.net/winhex/manual.pdf.

[12] Kristina Foster. Thesis: Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus. 2012. Last visited 2016-04-21 at http://simson.net/ref/2012/kmf_thesis.pdf.

[13] Simson Garfinkel, Alex Nelson, Douglas White, and Vassil Roussev. Using purpose-built functions and block hashes to enable small block and sub-file forensics. *Elsevier*, 2010. Last visited 2016-06-23 at http://www.sciencedirect.com/science/article/pii/S1742287610000307.

[14] Simson L. Garfinkel and Michael McCarrin. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Elsevier*, 2015. Last visited 2016-04-21 at http://www.sciencedirect.com/science/article/pii/S1742287615000468.

[15] Simson L. Garfinkel, Joel Young, Kristina Foster, and Kevin Fairbanks. Distinct sector hashes for target file detection. *Computer, 2012*, 2012. Last visited 2016-04-21 at http://ieeexplore.ieee.org.ucd.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=6311397.

[16] Angel Garrido. Classifying entropy measures. *Symmetry*, 2011. Last visited 2016-04-21 at http://www.mdpi.com/2073-8994/3/3/487/pdf.

[17] Yinghua Guo, Jill Slay, and Jason Beckett. Validation and verification of computer forensic software tools - searching function. *ScienceDirect, Elsevier*, 2009. Last visited 2016-06-08 at https://www.dfrws.org/2009/proceedings/p12-guo.pdf.

[18] Tony Hansen and Garrett Wollmann. Rfc 3174, us secure hash algorithm 1 (sha1). 2001. Last visited 2016-04-27 at https://tools.ietf.org/html/rfc3174.

[19] Dustin Hurlbut. Fuzzy hashing for digital forensic investigators. 2009. Last visited 2016-04-21 at https://ad-pdf.s3.amazonaws.com/Fuzzy_Hashing_for_Investigators.pdf.

[20] Andrey Jivsov. The use of secure hash algorithm 3 in openpgp (draft, expires feb 2016). 2015. Last visited 2016-04-27 at https://tools.ietf.org/html/draft-jivsov-openpgp-sha3-01.

[21] Simon Key. File block hash map analysis. *Guidance Software App Central*, 2013. Last visited 2016-04-21 at https://www2.guidancesoftware.com/appcentral/Pages/product.aspx?cat=GuidanceSoftware&pid=180010074WS.

[22] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Science Direct, Elsevier*, 2006. Last visited 2016-04-21 at http://dfrws.org/2006/proceedings/12-Kornblum.pdf.

[23] Kailash Kumar, Sanjeev Sofat, S. K. Jain, and Naveen Aggarwal. Significance of hash value generation in digital forensic: A case study. *International Journal of Engineering Research and Development*, 2012. Last visited 2016-06-08 at http://www.ijerd.com/paper/vol2-issue5/I02056470.pdf.

[24] Yuping Li, Xinming Ou, Sathya Chandran Sundaramurthy, Doina Caragea, Jiyong Jang, Alexandru G. Bardas, and Xin Hu. Experimental study of fuzzy hashing in malware clustering analysis. *USENIX*, 2015. Last visited 2016-04-21 at https://www.usenix.org/conference/cset15/workshop-program/presentation/li.

[25] Emily Namey, Greg Guest, Lucy Thairu, and Laura Johnson. Data reduction techniques for large qualitative data sets. 2007. Last visited 2016-04-27 at http://web.stanford.edu/~thairu/07_184.Guest.1sts.pdf.

[26] Alexander Noè. Avi file format. 2006. Last visited 2016-06-08 at http://www.alexander-noe.com/video/documentation/avi.pdf.

[27] Krzysztof Okupski. Bitcoin developer reference 2015. ., 2015. Last visited 2016-04-21 at http://enetium.com/resources/Bitcoin.pdf.

[28] Ronald. L. Rivest. Rfc 1321, the md5 message digest algorithm. 1992. Last visited 2016-04-27 at https://www.ietf.org/rfc/rfc1321.txt.

[29] Richard P. Salgado. Fourth amendment search and the power of the hash. 2005. Last visited 2016-05-24 at http://federalevidence.com/pdf/2013/02Feb/EE-4thAmSearch-Power%20of%20Hash.pdf.

[30] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal, Vol 27*, 1948. Last visited 2016-04-21 at http://www.essrl.wustl.edu/~jao/itrg/shannon.pdf.

[31] Matthew M. Shannon. Forensic relative strength scoring: Ascii and entropy scoring. *International Journal of Digital Evidence, Vol 2, Issue 4*, 2004. Last visited 2016-04-21 at http://digital4nzics.com/Student%20Library/Forensic%20Relative%20Strength%20Scoring-%20ASCII%20and%20Entropy%20Scoring.pdf.

[32] Sean Turner. Rfc 5754, using sha2 algorithms with cryptographic message syntax. 2010. Last visited 2016-04-27 at https://tools.ietf.org/html/rfc5754.

[33] Sriram Vajapeyam. Understanding shannon's entropy metric for information. 2014. Last visited 2016-04-21 at http://arxiv.org/pdf/1405.2061.pdf.

[34] Werner Vogel. File system usage in windows nt 4.0. *Symposium on Operating Systems Principles*, 1999. Last visited 2016-06-16 at https://www.cs.cornell.edu/projects/quicksilver/public_pdfs/File%20System%20Usage.pdf.

[35] Xiaoyun Wang and Songbo Yu. How to break md5 and other hash functions. 2005. Last visited 2016-04-21 at http://merlot.usc.edu/csac-f06/papers/Wang05a.pdf.

# Part V

# Appendices

# A
# Scripts

## A.1 Python Scripts

In this chapter we will include the most important scripts used in creating the datasets used for the different testing scenarios.

### A.1.1 Python script to create filehashes

Code A.1: Python script Filehashing_0.0.6.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:   /Users/Datakrim/Qsync/PHS/UCD/UCD-2014/Pythoncode/Filehashing_0.0.6.py
# Author:   Kurt H Hansen
# Created:  27.01.2015
# Modified: 17.01.2016
# Purpose:  MSc project. Create a database of blockhashes ripped from certain files
# Version:  0.0.6
#
# Changes   10.02.15 0.0.5  In main() a bug is fixed causing the
#                           hashing to start over if choosing
#                           both export to db and txt.That result
#                           in overwriting the exported text files.
#
# Planned   - Add a function to test if the output textfile if
#             dbTXT=True exists and if you suppose to overwrite it.
#           - Find possibilities to speed up the processes


import sys, os, math, hashlib, datetime, time
import mysql.connector      # The MySQL engine
from mysql.connector import errorcode

reload(sys)
sys.setdefaultencoding("latin-1")

########## Static values ######################
global FileTypesToEvaluate
global PrintToTXT
global PrintToDB

PrintToTXT = True
PrintToDB = False
# The path were to hash from
FilesToHashPath = r'/Volumes/Rugged_Key 1/'
# Give all in lower-case. The search convert the filename to lc.
#FileTypesToEvaluate = ('avi', 'mpg', 'mpeg', 'mov', 'wmv', 'mp4', 'm4p',
#                        'm4v','sub','divx','rmvb','flv','ts','vob','mkv')

# Give all in lower-case. The search convert the filename to lc.
FileTypesToEvaluate = ('jpg')
# The file path+name and MD5 is dumped to this file if PrintToTXT = True
FileToDumpTSV = "/Volumes/Rugged_Key 1/GoProTSV.txt"

#####################################################################
# The MainFileHashing function do the following:
# - Traverse a given folder and its subfolder
# - Creates a MD5 hash of files with ending found in the FileTypesToEvaluate
# - Save the file-path+name and MD5 to MySQL and/or TSV file
#
#####################################################################
def MainFileHashing(cnx, cursor):
    TotalTimeStart=time.time()
    TotalFileSize=0
    # Opens the output TSV file for write
    FilehashTSV=open(FileToDumpTSV, "w")
    # Set the searchdir static
    for path, subdirs, files in os.walk(FilesToHashPath):
        for filename in files:
            # Make a lower-case instance of the filename when
            filenameTOlower=filename.lower()
            # in next line check for valid suffixes in lower-case
            # Check if the file endings are according to the
            if filenameTOlower.endswith(FileTypesToEvaluate):
                # global variable set of extensions
                FileTimeStart=time.time()
                # Concatenate File path + name
                FilePathName = os.path.join(path, filename)
                # Open the file, read only, binary
                f1=open(FilePathName,"rb").read()
                # Calculate the MD5 of the file
                MD5hash=hashlib.md5(f1).hexdigest()
                fsize=os.path.getsize(FilePathName)
                fext=os.path.splitext(FilePathName)[1][1:].lower()
                if PrintToTXT:
                    # Write to the textfile if PrintToTXT = True
                    FilehashTSV.write(str(FilePathName)+"\t"+ MD5hash + "\t" + \
                                      str(fsize) + "\t" + fext + os.linesep)
                if PrintToDB:
                    # Write to MySQL db if PrintToDB = True
                    dbWriteRecord_Filehash(cursor, [FilePathName,MD5hash, fsize, fext])
                    cnx.commit()
                FileTimeEnd = time.time()
                TotalFileSize += os.path.getsize(FilePathName)
                print "Finished in", FileTimeEnd-FileTimeStart, " s, Size: ", \
                        ("{:,}".format(os.path.getsize(FilePathName))), " File:",
                            FilePathName
    FilehashTSV.close()
    TotalTimeEnd=time.time()
    print "Total processing time=", TotalTimeEnd  - TotalTimeStart, " Seconds"
    print "Total bytes processed:", ("{:,}".format(TotalFileSize))," Bytes"
    return True


######################################################################
# The dbWriteRecord_Filehash function do the following:
# - Receives the cursor object and the values to store in the
# - table Hashdatabase in the database MSc
# - The Filename and MD5 is written to the MSc.Hashdatabase
#
######################################################################
def dbWriteRecord_Filehash(cursor, dbRecord):
    # dbRecord is CursorObject, [FileID, sha256hash, blocknum, Entropy]

    dbAddBlockhash = ("INSERT INTO hashdatabase "
                     "(Filename, MD5, Filesize, Filetype)"
                     "VALUES (%s, %s, %s, %s)"
                    )
    dataAddBlockhash =(dbRecord[0],
                       dbRecord[1],
                       dbRecord[2],
                      dbRecord[3]
                      )
    cursor.execute(dbAddBlockhash, dataAddBlockhash)
    return True

def dbCreateTable(cnx, cursor):
    ################################################
    # Define the tables in a dictionary of table defs
    ################################################
    TABLES={}
    TABLES['hashdatabase'] = (
        "CREATE TABLE hashdatabase ("
        " id int(11) NOT NULL AUTO_INCREMENT,"
        " Filename varchar(255) NOT NULL,"
        " MD5 varchar(32) NOT NULL,"
        " Filesize bigint(20),"
        " Filetype varchar(12),"
        " Blockhashed varchar(15),"
        #" INDEX Hash USING BTREE (MD5(8)), "
        "  PRIMARY KEY (id)"
        ") ENGINE=MYISAM" )


    ################################################
    # Iter trough the dictionary and execute the creation
    # of tables trough the cursor object
    ################################################
    for name, ddl in TABLES.iteritems():
        try:
            print('Creating table {}: '.format(name))
            cursor.execute(ddl)
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
                print("The table already exists.")
                #########################################################
                # Shall we erase the existing table or not.
                # If not, the new hashes is appended to the table
                #########################################################

                isDropTable=raw_input("The table " + name + \
                                      " exists. Do you want to delete the existing \
                                      and create new table (yes/no): ")
                if isDropTable.lower() == 'yes':
                    sql = "DROP TABLE IF EXISTS %s" % (name)
                    cursor.execute(sql)             # Drops the Table Hashdatabase
                    cnx.commit()
                    print "The existing table {} is dropped".format(name)
                    cursor.execute(ddl)             # Creates a new empty Hashdatabase
                    cnx.commit()
                    print "The table {} is created and empty, "+ \
                        "ready for refueling".format(name)
            else:
                print(err.msg)
        else:
            print("OK")

    return True


############## Main ##################################
#
# Use the Oracle mySQL plugin to handle mySQL databases
# Open te database MSc
#
#######################################################
def main():

    if PrintToDB:
        try:
            cnx =mysql.connector.connect(user='root', host='localhost', database='msc')
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
                print("Something is wrong with your user name or password")
            elif err.errno == errorcode.ER_BAD_DB_ERROR:
                print("Database does not exists")
            else:
                print(err)
        else:
            print("Connected to the database successfully !")

        cursor=cnx.cursor()

        dbCreateTable(cnx, cursor)

        #######################################################
        # Starts the following:
        # MainFileHashing(cursor)       # The Hashing engine
        #######################################################
        BlockHashingResult = MainFileHashing(cnx, cursor)
        cursor.close()
        cnx.commit()
        cnx.close()
    else:
        BlockHashingResult = MainFileHashing(False,False)
    return True


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

### A.1.2 Python script to create blockhashes

Code A.2: Python script BlockHashing_0.1.3.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:   /Users/Datakrim/Qsync/PHS/UCD/UCD-2014/Pythoncode/BlockHashing_0.1.3.py
# Author:   Kurt H Hansen
# Created:  27.01.2015
# Modified: 16.01.2016
# Purpose:  MSc project. Create a database of blockhashes ripped from certain files
# Version:  0.1.3
# Changes:  09.02.15 0.1.1  All references to database and tables is now in lowercase
    letters
#                           like msc, blockhash* and filenames*
#           09.02.15 0.1.2  Changes the fields exported/written to db.
#                           Not use path and file type any more
#           16.01.16 0.1.3  Improve the documentation and the tables at the bottom
#
import sys, os, math, struct, hashlib, time
import entropy                              # https://pypi.python.org/pypi/entropy/0.9

import encodings
import unicodedata
import mysql.connector                      # The MySQL engine
from mysql.connector import errorcode

reload(sys)
sys.setdefaultencoding("utf8")

def usage():
    print len(sys.argv)
    print "The script was started wtih  wrong options"
    print "The script require two arguments !"
    print "Use the following syntax: "
    print sys.argv[0], " <Full path to pictures> <full path/filename to TSV file> <file ext
        >"
    sys.exit()

#-----Global static values --------------------------------------------------
global InitialBlockSize
global HashType        # Not yet in use
global PrintToTXT
global PrintToDB

InitialBlockSize = 8192
HashType="SHA256"
PrintToTXT = True           # True = Dumps the records to a textfile
PrintToDB = False           # True = Dumps the records to MySQL database
PrintToTXTDelimiter=','

# The path were to hash from
FilesToHashPath = r'/Volumes/LaCie/msc_pictures/'
# The file path+name and MD5 is dumped to this file if PrintToTXT = True
FileBlockhashToDumpTSV = "/Volumes/LaCie/msc_pictures/Blockhash" + str(InitialBlockSize)+".
    txt"
# The file path+name and MD5 is dumped to this file if PrintToTXT = True
FileNamesToDumpTSV = "/Volumes/LaCie/msc_pictures/Filenames" + str(InitialBlockSize)+".txt"

#####################################################################
# The BlockHashing function do the following:
# - Traverse a given folder and its subfolder
# - Divide the file into blocks os size = InitialBlockSize
# - Calculates the SHA256 of the block
# - Calculates the entropy of the block
# - Ommit the last block of a fileif size < InitialBlockSize
#
#
#####################################################################
def BlockHashing(cnx, cursor):
    '''
    ------------------------------------------------------------------------
    Opens the msc --> hashdatabase
    This is the database of files already hashed and ensured no duplicates
    This is the connection to read file values to traverse
    ------------------------------------------------------------------------
```

```
'''
try:
    cnx2=DatabaseConnection()
    # cnx2 =mysql.connector.connect(user='root', host='localhost',
    # password='root', database='msc_pictures')
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your user name or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exists")
    else:
        print(err)
else:
    print("Connected to the database successfully, \
          connection used for the Hashdatabase!")

cursor2=cnx2.cursor()

'''
-----------------------------------------------------------------------
Opens the msc --> hashdatabase
This is the database of files already hashed and ensured no duplicates
This is the connection to make update to Hashdatabase
-----------------------------------------------------------------------
'''
try:
    cnx3=DatabaseConnection()
    # cnx3 =mysql.connector.connect(user='root', password='root',
    # host='localhost', database='msc_pictures')
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your user name or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exists")
    else:
        print(err)
else:
    print("Connected to the database successfully, \
          connection used for the hashdatabase!")

cursor3=cnx3.cursor()

'''
-----------------------------------------------------------------------
Select the database of files that is hashed for the dataset
-----------------------------------------------------------------------
'''
cursor2.execute("SELECT id, Filename, Filetype, Blockhashed  \
                FROM hashdatabase where id > 0")


TotalTimeStart=time.time()                  # Timer, Total
TotalFileSize=0                             # Sumarize file sizes
FileID = 0                                  # The FileID is the uniq number to
                                            # connect filenmes to Blockhashes
FileRemnantsExists = False
if PrintToTXT:
    a=open(FileNamesToDumpTSV, "w")         # The file where to dump filenames (Use "a"
         to append)
    b=open(FileBlockhashToDumpTSV, "w")     # The file where to dump Block Hashes


row = cursor2.fetchone()                    # Retrieves the first file from msc-->
    hashdatabase
while row is not None:                     # row[0]=id row[1]=Filename
                                            # row[2]=Filetype row[3]=Blockhashed

    FileTimeStart=time.time()                           # Timer start on each file
    f_status=os.path.isfile(row[1])                     # Check if the file still
        exists
    FilePathName = row[1]                               # Get file pat+name from
        Hashdatabase
    FileNameExt=os.path.splitext(FilePathName)[1][1:]   # Pick the file name extension,
         without the .
    FileID = row[0]                                     # File number
```

```
        # Determine the file size
        filelength=os.path.getsize(FilePathName)
        # Check if the last part of the file is < InitialBlockSize
        if filelength % InitialBlockSize > 0:
            FileRemnantsExists = 1
        else:
            FileRemnantsExists = 0
        if PrintToTXT:
            a.write(str(FileID) + PrintToTXTDelimiter + str(FileRemnantsExists) + os.
                linesep)
        if PrintToDB:
            dbWriteRecord_Filename(cursor, [FileID, FileRemnantsExists])

        '''
        ------------------------------------------------------------------------
        Open the selected file. Read all the content in one chunk
        The Blockhashing starts here
        ------------------------------------------------------------------------
        '''
        # Open the file, read only, binary
        f1=open(FilePathName,"rb").read()
        blocknum=0
        for i in range(0, filelength,InitialBlockSize):
            block=f1[i:i+InitialBlockSize]
            # If the block size < 512 bytes, ommit it.
            if len(block) < InitialBlockSize:
                # Indicates if True the file has a chunk at the end
                # not processed because < InitialBlockSize
                FileRemnantsExists = True
            else:
                # Calculate the SHA256 of the block
                sha256hash=hashlib.sha256(block).hexdigest()
                # Calculates the entropy of the block Entropy = 0-1
                Entropy = entropy.shannon_entropy(block)
                if PrintToTXT:
                    b.write(str(FileID) + PrintToTXTDelimiter + sha256hash+
                        PrintToTXTDelimiter + \
                            str(blocknum) + PrintToTXTDelimiter + str(Entropy) + os.linesep
                                )
                if PrintToDB:
                    dbWriteRecord_Blockhash(cursor, [FileID,
                                                        sha256hash,
                                                        blocknum,
                                                        Entropy
                                                        ])
                FileTimeEnd = time.time()
                blocknum +=1

        TotalFileSize += os.path.getsize(FilePathName)
        blocknum=0                                  # File is processed, set block to 0
        FileRemnantsExists=False                    # Resets the indicator of smal ending
            of file
        if PrintToDB:
            cnx.commit()
        tmp_fsize=os.path.getsize(FilePathName)
        print "ID:", FileID, " Time:", round(FileTimeEnd-FileTimeStart,2), "s, Speed:", \
            round(tmp_fsize/(FileTimeEnd-FileTimeStart),3), " b/s Size: ", \
            ("{:,}".format(tmp_fsize)), " File:", FilePathName
        '''
        ------------------------------------------------------------------------
        Evaluating the existing value in Hashdatabase-->Blockhashed
        If it contains the value from InitialBlockSize, we need not to change
        If the value not exists, the string is added, eks. '512:'
        The format of the field is by example '512:1024:4096:'
        All 3 indicates the file is blockhashed for the 3 sizes of blockhash
        ------------------------------------------------------------------------
        '''
        Hashdatabase_InitialBlockSize=row[3]
        str_search = -1
        if isinstance(Hashdatabase_InitialBlockSize, basestring):
            str_search=Hashdatabase_InitialBlockSize.find(str(InitialBlockSize)+':')
        else:
            Hashdatabase_InitialBlockSize = ''
        if str_search < 0:      # Negative value indicates that the string is not found in
            the string
            Hashdatabase_InitialBlockSize += str(InitialBlockSize) + ':'
            sql='UPDATE Hashdatabase SET Blockhashed="%s" \
                WHERE id=%s' % (Hashdatabase_InitialBlockSize, row[0])
            #print sql
            cursor3.execute(sql)
            cnx3.commit()
        row = cursor2.fetchone()        # Retrieves the next file from msc-->hashdatabase

    '''
    -----------------------------------------------------------------------
    The Blockhashing ends here per file
    -----------------------------------------------------------------------
    '''
    if PrintToTXT:
        b.close()
        a.close()
    TotalTimeEnd=time.time()
    print "Total processing time=", TotalTimeEnd  - TotalTimeStart, " Seconds"
    print "Total bytes processed:", ("{:,}".format(TotalFileSize))," Bytes"
    print "IO speed in b/s:", ("{:,}".format(TotalFileSize/(TotalTimeEnd  - TotalTimeStart)
        ))," Bytes"
    '''
    -----------------------------------------------------------------------
    Closes the cnx3 connection to msc-->hashdatabase    Read filedata
    -----------------------------------------------------------------------
    '''
    cursor3.close()
    cnx3.commit()
    cnx3.close()
    '''
    -----------------------------------------------------------------------
    Closes the cnx2 connection to msc-->hashdatabase    Write filedata
    -----------------------------------------------------------------------
    '''
    cursor2.close()
    cnx2.commit()
    cnx2.close()
    return True

#####################################################################
# The dbWriteRecord_Filename function do the following:
# - Receives the cursor object and the values to store in two database tables
# - The FileID and FilePathName , FileType and FileRemnant is written to the msc.filenames
#
#####################################################################
def dbWriteRecord_Filename(cursor, dbRecord):

    dbAddFilename = ("INSERT INTO filenames"+str(InitialBlockSize)+" "
                     "(FileNum, Filename, Filetype, FileRemnant)"
                     "VALUES (%s, %s, %s, %s)"
                    )
    dataAddFilename =(dbRecord[0],
                      dbRecord[1],
                      dbRecord[2],
                      dbRecord[3]
                      )
    cursor.execute(dbAddFilename, dataAddFilename)

    return True

#####################################################################
# The dbWriteRecord_Blockhash function do the following:
# - Receives the cursor object and the values to store in two database tables
# - The FileID and FilePathName , FileType and FileRemnant is written to the msc.filenames
# - The FileID, sha256hash, blocknum, Entropy is written to the msc.blockhash
#
#####################################################################
def dbWriteRecord_Blockhash(cursor, dbRecord):

    dbAddBlockhash = ("INSERT INTO blockhash"+str(InitialBlockSize)+" "
                     "(FileNum, SHA256, BlockNum, Entropy)"
                     "VALUES (%s, %s, %s, %s)"
                    )
    dataAddBlockhash =(dbRecord[0],
                      dbRecord[1],
                      dbRecord[2],
                      dbRecord[3]
```

```
                       )
    cursor.execute(dbAddBlockhash, dataAddBlockhash)
    return True

def dbCreateTables(cursor):
    '''
    -----------------------------------------------------------------------
    # Define the tables in a dictionary of table defs
    -----------------------------------------------------------------------
    '''
    TABLES={}

    TBL_Filenames="filenames"+str(InitialBlockSize)
    TABLES[TBL_Filenames] = (
        "CREATE TABLE "+TBL_Filenames+"("
        " FileNum int(11) NOT NULL,"
        " FileRemnant boolean default 0,"
        "  PRIMARY KEY (FileNum)"
        ") ENGINE=MYISAM" )


    TBL_Blockhash="blockhash"+str(InitialBlockSize)
    TABLES[TBL_Blockhash] = (
        "CREATE TABLE "+TBL_Blockhash+" ("
        " id bigint(24) NOT NULL AUTO_INCREMENT,"
        " FileNum int(11) NOT NULL,"
        " SHA256 varchar(64) NOT NULL,"
        " BlockNum int(11) NOT NULL,"
        " Entropy double NOT NULL,"
        #" INDEX Hash USING BTREE (SHA256(8)), "
        "  PRIMARY KEY (id)"
        ") ENGINE=MYISAM" )

    '''
    -----------------------------------------------------------------------
    # Iter trough the dictionary and execute the creation
    # of tables trough the cursor object
    -----------------------------------------------------------------------
    '''
    for name, ddl in TABLES.iteritems():
        try:
            print('Creating table {}: '.format(name))
            cursor.execute(ddl)
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
                print("The table already exists.")
            else:
                print(err.msg)
        else:
            print("OK")
    return True
'''
-----------------------------------------------------------------------
# Use the Oracle mySQL plugin to handle mySQL databases
# Open te database msc
-----------------------------------------------------------------------
'''
def main():
    # Make connection if decided to dump data directly to the MySQL database.
    # Check the global varible in the beginning of the script
    # Preferable, dump the data to textfile, later import to MySQL with
    # LOAD DATA INFILE ...
    #
    if PrintToDB:
        # Make connection to database
        try:
            cnx=DatabaseConnection()
            # cnx =mysql.connector.connect(user='root', host='localhost',
            # password='root', database='msc_pictures')
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
                print("Something is wrong with your user name or password")
            elif err.errno == errorcode.ER_BAD_DB_ERROR:
                print("Database does not exists")
            else:
                print(err)
        else:
            print("Connected to the database successfully, connection \
                  used for Blockhash and filenames !")

        cursor=cnx.cursor()
        dbCreateTables(cursor)
        '''
        -----------------------------------------------------------------------
        # Starts the following:
        # Blockhshing(cursor)       # The blockhashing w/Entropy
        -----------------------------------------------------------------------
        '''
        BlockHashingResult = BlockHashing(cnx, cursor)
        cursor.close()
        cnx.commit()
        cnx.close()
    else:
        # Prepare to not dump to database but to textfile
        BlockHashingResult = BlockHashing(False, False)

    return BlockHashingResult

def DatabaseConnection():
    return mysql.connector.connect(user='root', password='root', \
                                   host='localhost', database='msc_pictures')

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

### A.1.3 Python script to blockhash the case

Code A.3: BlockHashingCase_0.0.1.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:   /Users/Datakrim/Dropbox/PHS/UCD/UCD-2014/Pythoncode/BlockHashingCase_0.0.1.py
# Author:   Kurt H Hansen
# Created:  13.02.2015
# Modified:
# Purpose:  MSc project. Create a database of blockhashes ripped from case file 8916796
# Version:  0.0.1
#
#
import sys, os, math, struct, hashlib, time
import entropy                                  # https://pypi.python.org/pypi/entropy/0.9

InitialBlockSize = 512
HashType="SHA256"
PrintToTXTDelimiter=','

# The file were to hash from
FilesToHashPath = r'/Volumes/msc_khh/891679616796_2013_1732_A5.dd'

# The file path+name and MD5 is dumped to this file if PrintToTXT = True
FileBlockhashToDumpTSV = "/Volumes/LaCie/msc_case_dbTXT/Blockhash" + str(InitialBlockSize)+
    ".txt"

#####################################################################
# The BlockHashing function do the following:
# - Divide the file into blocks os size = InitialBlockSize
# - Calculates the SHA256 of the block
# - Calculates the entropy of the block
# - Ommit the last block of a fileif size < InitialBlockSize
#
#
#####################################################################
def main():

    TotalTimeStart=time.time()      # Timer, Total
    FileRemnantsExists = False

    DumpFile=open(FileBlockhashToDumpTSV, "w")      # The file where to dump Block Hashes

    '''
    ----------------------------------------------------------------------
    Open the selected file. Read the content block by block
    ----------------------------------------------------------------------
    '''
    f1=open(FilesToHashPath,"rb")                               # Open the file, read only,
        binar
    filelength=os.path.getsize(FilesToHashPath)                 # Determine the file size
    TotalBlocks=int(filelength/InitialBlockSize)
    blocknum=0
    for i in range(0, filelength,InitialBlockSize):
        block=f1.read(InitialBlockSize)
        if len(block) < InitialBlockSize:       # If the block size < 512 bytes, ommit it.
            FileRemnantsExists = True           # Indicates if True the file has a chunk at
                 the
                                                # end not processed because <
                                                    InitialBlockSize
        else:
            # Calculate the SHA256 of the block
            sha256hash=hashlib.sha256(block).hexdigest()
            # Calculates the entropy of the block Entropy = 0-1
            Entropy = entropy.shannon_entropy(block)
            DumpFile.write(sha256hash+PrintToTXTDelimiter + str(blocknum) + \
                           PrintToTXTDelimiter + str(Entropy) + os.linesep)
            FileTimeEnd = time.time()
            blocknum +=1
            if blocknum % 100000 == 0:
                print "Blocks processed so far: ", ('{:,}').format(blocknum)

    '''
    ----------------------------------------------------------------------
    The Blockhashing ends here per file
    ----------------------------------------------------------------------
    '''
    DumpFile.close()
    f1.close()
    TotalTimeEnd=time.time()
    print "Total processing time=", TotalTimeEnd  - TotalTimeStart, " Seconds"
    print "Total bytes processed:", ("{:,}".format(filelength))," Bytes"
    print "IO speed in b/s:", ("{:,}".format(TotalFileSize/(TotalTimeEnd  - TotalTimeStart)
        ))," Bytes"
    print "Last block incomplete: ",FileRemnantsExists
    return True

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

### A.1.4 Python script to remove duplicate files in video database

Code A.4: Duplicates_Remove_0.0.2.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:   /Users/Datakrim/Dropbox/PHS/UCD/UCD-2014/Pythoncode/Duplicates_Remove_0.0.1.py
# Author:   Kurt H Hansen
# Created:  31.01.15
# Modified: 09.02.15
# Purpose:  Reads data from Msc.Hashdatabase
#           Pick unique records based on MD5. Remove all duplicate file both
#           in database and the actual file
#           If by example there is 4 files with same MD5, 3 is deleted
#
#           By setting the variable createBash=True, the duplicate files is not
#           directly erased but the rm <<pat/file>> per file is put in the
#           Bash file: EraseDuplicateFiles.sh
# Version:  0.0.2
# Depends:  Running the following script prior to this to create the database of MD5 hashes
     of files:
#           /Users/Datakrim/Dropbox/PHS/UCD/UCD-2014/Pythoncode/Filehashing_0.0.2.py
# Changes:      0.0.2   All references to database and tables are now in lowercase letters
#                       Cleaned up in the script

import sys, os, dircache
import encodings
import unicodedata
reload(sys)
sys.setdefaultencoding("utf8")

import mysql.connector

# Set this value to True if no direct erase of duplicate
# files but store the erase command in a bash script
createBash=True

db=mysql.connector.connect(host="localhost", user="root", db="msc")
cur=db.cursor()

print "Executes statement on Hashdatabase to pick duplicate \
        records based on hash. This could take time ...."
#
# The following SQL-query picks records with equal MD5.
# All records with more than 2 MD5 with same value
#
cur.execute("select id,Filename,hashdatabase.MD5 from hashdatabase inner join \
            (select MD5 from hashdatabase group by MD5 having count(id) > 1) \
            dup on hashdatabase.MD5 = dup.MD5")
print "The SQL statement on database Hashdatabase is finished ..."
print "Finding are evaluated ..."

tmpMD5=''
if createBash:
        a=open("EraseDuplicateFiles.sh","w")     # The bash script file open for write
        a.write("#!/bin/bash" + os.linesep)      # Initialize the bash script  shebang
for row in cur.fetchall() :                      # Traverse trough the whole dataset
    f_MD5=row[2]
    if f_MD5 == tmpMD5:
        f_name=row[1]
        f_status=os.path.isfile(f_name)          # Check if file defined in database exists
        f_id=row[0]
        if f_status:
            # Routine for erasing the physical file
            # as we now have identified the file as
            # a copy and the file actually exists on the disk
            #
            # 1. Erase the file:
            #
            try:
                if createBash:
                    # Write to the EraseDuplicateFiles.sh bash file
                    a.write("rm " + '"' + f_name + '"' + os.linesep)
                else:
                    # Erases the file given from record in the Hashdatabase
                    os.remove(f_name)
            ## if failed, report it back to the user ##
            except OSError, e:
                print ("Error: %s - %s." % (e.filename,e.strerror))
            print "We have erased: ", f_name, " Status=",f_status
        else:
            print "The file: ", f_name, " does not exist ...."

        # Next, the duplicate record in the Hasdatabase will be removed
        # 2. Erase the record
        # The record(s) will be erased even if the corresponding file does not exist
        #
        sql_statmt=("DELETE FROM %s WHERE id = %s") %('hashdatabase', f_id)
        #print sql_statmt
        cur.execute(sql_statmt)     # Delete the requested record in Hashdatabase
        db.commit()
    tmpMD5 = f_MD5
a.close()
db.close()
```

### A.1.5 Python script to calculate average color of the jpg pictures

Code A.5: Pictures_Pixelcalc_0.0.4.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:       /Users/Datakrim/Qsync/PHS/UCD/UCD-2014/Pythoncode/Pictures_Pixelcalc_0.0.4.
    py
# Author:       Kurt H Hansen
# Created:      19.02.2015
# Modified:     16.01.2016
# Purpose:      MSc to calculate the color average in 40500 pictures
# Version:      0.0.4
# Changes:      Addes sys.argv with 3 arguments.
#               Replaced the picture calculation engine
# Example       python Pictures_Pixelcalc_0.0.4.py ./Bilder/ ./Pixelcalc.txt jpg
# Links:        http://blog.iconfinder.com/detecting-duplicate-images-using-python/
# Credit        http://pythonicprose.blogspot.no/2009/09/python-find-average-rgb-color-for-
    image.html
#               Steve ??
# Resource      http://rapidtables.com/convert/color/index.htm

import sys, os, math, datetime, time
from PIL import Image

#
# Checking the startup of the script that requires two arguments, separated with <space>
#
def usage():
    print len(sys.argv)
    print "The script was started wtih  wrong options"
    print "The script require two arguments !"
    print "Use the following syntax: "
    print sys.argv[0], " <Full path to pictures> <full path and filename \
                        to TSV file> <file extension>"
    sys.exit()


class PixelCounter(object):
  ''' loop through each pixel and average rgb '''
  def __init__(self, imageName):
      self.pic = Image.open(imageName)
      # load image data
      self.imgData = self.pic.load()
  def averagePixels(self):
      r, g, b = 0, 0, 0
      count = 0
      for x in xrange(self.pic.size[0]):
          for y in xrange(self.pic.size[1]):
              tempr,tempg,tempb = self.imgData[x,y]
              r += tempr
              g += tempg
              b += tempb
              count += 1
      # calculate averages
      PixelAverage=(r/count * 256**2)+(g/count * 256**1)+(b/count * 256**0)

      return (r/count), (g/count), (b/count), count,PixelAverage


######################################################################
# The color calculation function do the following:
# - Traverse a given folder and its subfolder
# - Clculate the average color (rgb pairs, decimal and Hex)
# - Export the calculations to a TSV file
#
######################################################################
def MainPictureCalculation(PathToPictures, FileToDumpTSV, FileTypesToEvaluate):
     TotalTimeStart=time.time()
     # Opens the output TSV file for write
     OutputTxtCSV=open(FileToDumpTSV, "w")
     # Set the searchdir static
     for path, subdirs, files in os.walk(PathToPictures):
         for filename in files:
           t1=time.time()
           # Make a lower-case instance of the filename when in next line
```

```
            # check for valid suffixes in lower-case
            filenameTOlower=filename.lower()
            # Check if the file endings are according to the global
            # variable set of extensions
            if filenameTOlower.endswith(FileTypesToEvaluate):
                 FileTimeStart=time.time()
                 # Concatenate File path + name
                 FilePathName = os.path.join(path, filename)

                 pc = PixelCounter(FilePathName)
                 #print "(red, green, blue, total_pixel_count, average color decimal)"
                 PictureArray= pc.averagePixels()

                 ColorHex=hex(PictureArray[4]).rstrip("L").lstrip("0x") or "0"
                 ColorR = PictureArray[0]
                 ColorG = PictureArray[1]
                 ColorB = PictureArray[2]
                 ColorRGB=(ColorR,ColorG,ColorB)
                 OutputTxtCSV.write(os.path.basename(FilePathName)+"\t"+ \
                                   str(PictureArray[4]) +"\t"+ \
                                   ColorHex + "\t"+ str(ColorRGB) + os.linesep)
                 t2 = time.time()
                 print "Finished in", round(t2-t1,2), "sec, Size: ", \
                         ("{:,}".format(os.path.getsize(FilePathName))), \
                         " File:", os.path.basename(FilePathName)
     OutputTxtCSV.close()
     TotalTimeEnd=time.time()
     print "Total processing time=", TotalTimeEnd  - TotalTimeStart, " Seconds"
     return True

# ---------- Main part of code. Execution starts here ----------
if __name__ == '__main__':

     if (len(sys.argv) < 4):         # Checks that the full path is given and the log-file
          usage()
     print sys.argv
     MainPictureCalculation(sys.argv[1], sys.argv[2], sys.argv[3])
     print "Finished ..."
```

### A.1.6 Python script to convert JPG pictures to BMP-2

Code A.6: ConvertJPGtoBMP2.py

```
#!/usr/bin/env python

# From: ActiveState
# http://code.activestate.com/recipes/180801-convert-image-format/
"""Program for converting image files from one format
to another. Will convert one file at a time or all
files (of a selected format) in a directory at once.
Converted files have same basename as original files.

Uses workaround: askdirectory() does not allow choosing
a new dir, so asksaveasfilename() is used instead, and
the filename is discarded, keeping just the directory.
"""
import os, os.path, string, sys
from Tkinter import *
from tkFileDialog import *
from PIL import Image

openfile = '' # full pathname: dir(abs) + root + ext
indir = ''
outdir = ''
def getinfilename():
    global openfile, indir
    ftypes=(('Gif Images', '*.gif'),
            ('Jpeg Images', '*.jpg'),
            ('Png Images', '*.png'),
            ('Tiff Images', '*.tif'),
            ('Bitmap Images', '*.bmp'),
            ("All files", "*"))
    if indir:
        openfile = askopenfilename(initialdir=indir,
                                   filetypes=ftypes)
    else:
        openfile = askopenfilename(filetypes=ftypes)
    if openfile:
        indir = os.path.dirname(openfile)

def getoutdirname():
    global indir, outdir
    if openfile:
        indir = os.path.dirname(openfile)
        outfile = asksaveasfilename(initialdir=indir,
                                    initialfile='foo')
    else:
        outfile = asksaveasfilename(initialfile='foo')
    outdir = os.path.dirname(outfile)

def save(infile, outfile):
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print "Cannot convert", infile

def convert():
    newext = frmt.get()
    path, file = os.path.split(openfile)
    base, ext = os.path.splitext(file)
    if var.get():
        ls = os.listdir(indir)
        filelist = []
        for f in ls:
            if os.path.splitext(f)[1] == ext:
                filelist.append(f)
    else:
        filelist = [file]
    for f in filelist:
        infile = os.path.join(indir, f)
        ofile = os.path.join(outdir, f)
        outfile = os.path.splitext(ofile)[0] + newext
        save(infile, outfile)
    win = Toplevel(root)
    Button(win, text='Done', command=win.destroy).pack()

# Divide GUI into 3 frames: top, mid, bot
root = Tk()
topframe = Frame(root,
                 borderwidth=2,
                 relief=GROOVE)
topframe.pack(padx=2, pady=2)

midframe = Frame(root,
                 borderwidth=2,
                 relief=GROOVE)
midframe.pack(padx=2, pady=2)

botframe = Frame(root)
botframe.pack()

Button(topframe,
       text='Select image to convert',
       command=getinfilename).pack(side=TOP, pady=4)

multitext = """Convert all image files
(of this format) in this folder?"""
var = IntVar()
chk = Checkbutton(topframe,
                  text=multitext,
                  variable=var).pack(pady=2)
Button(topframe,
       text='Select save location',
       command=getoutdirname).pack(side=BOTTOM, pady=4)


Label(midframe, text="New Format:").pack(side=LEFT)
frmt = StringVar()
formats = ['.bmp', '.gif', '.jpg', '.png', '.tif']
for item in formats:
    Radiobutton(midframe,
                text=item,
                variable=frmt,
                value=item).pack(anchor=NW)

Button(botframe, text='Convert', command=convert).pack(side=LEFT,
                                                       padx=5,
                                                       pady=5)
Button(botframe, text='Quit', command=root.quit).pack(side=RIGHT,
                                                      padx=5,
                                                      pady=5)

root.title('Image Converter')
root.mainloop()
```

### A.1.7 Python script to blockhash the BMP-2 files

Code A.7: BlockhashPicturesBMP_0.0.1.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:      /Users/Datakrim/Qsync/PHS/UCD/UCD-2014/
#               Pythoncode/BlockhashPicturesBMP_0.0.1.py
# Author:      Kurt H Hansen
# Created:     02.02.2016
# Modified:
# Purpose:     MSc to blockhash BMP2 files converted from JPG
# Version:     0.0.1
# Changes:
# Example

import sys, os, math, datetime, time
from PIL import Image
import entropy               # https://pypi.python.org/pypi/entropy/0.9
import hashlib

InitialBlockSize = 512
UpdateHashdatabase=False
HashType="SHA256"
# Give all in lower-case. The search convert the filename to lc.
FileTypesToEvaluate = ('bmp')
TotalFileSize = 0
PrintToTXTDelimiter=','


# The path were to hash from
FilesToHashPath = r'/Volumes/LaCie/msc_pictures_raw/'
# The file path+name and MD5 is dumped to this file if PrintToTXT = True
FileBlockhashToDumpCSV = "/Volumes/LaCie/msc_pictures_raw_txt/Blockhash" + \
                         str(InitialBlockSize)+".txt"

#####################################################################
# The color calculation function do the following:
# - Traverse a given folder and its subfolder
# - Create blockhash SHA256 and entropy of each block
# - Export the calculations to a CSV file
#   prepared for import into MySQL database
#
#####################################################################

TotalTimeStart=time.time()
OutputTxtCSV=open(FileBlockhashToDumpCSV, "a")          # Opens the output CSV file for
    write
for path, subdirs, files in os.walk(FilesToHashPath):   # Set the searchdir static
    for filename in files:
        t1=time.time()
        # Make a lower-case instance of the filename when in next
        # line check for valid suffixes in lower-case
        filenameTOlower=filename.lower()
        # Check if the file endings are according to the
        # global variable set of extensions
        if filenameTOlower.endswith(FileTypesToEvaluate) and int(filenameTOlower[3:-4]) >
            0:
            FileTimeStart=time.time()
            filelength=os.path.getsize(FilesToHashPath+filename)

            FileSkipBytes=54
            f1=open(FilesToHashPath+filename,"rb").read()
            blocknum=0
            #xxx=open(FilesToHashPath+filename+".DD","wb")
            #print FilesToHashPath+filename+".DD"
            for i in range(FileSkipBytes, filelength,InitialBlockSize):
                block=f1[i:i+InitialBlockSize]
                #xxx.write(block)
                # If the block size < 512 bytes, ommit it.
                if len(block) < InitialBlockSize:
                    # Indicates if True the file has a chunk at the
                    # end not processed because < InitialBlockSize
                    FileRemnantsExists = True
                else:
                    # Calculate the SHA256 of the block
                    sha256hash=hashlib.sha256(block).hexdigest()
                    # Calculates the entropy of the block Entropy = 0-1
                    Entropy = entropy.shannon_entropy(block)
                    OutputTxtCSV.write(str(filename) + PrintToTXTDelimiter + \
                                       sha256hash+PrintToTXTDelimiter + str(blocknum) + \
                                        PrintToTXTDelimiter + str(Entropy) + os.linesep)
                    FileTimeEnd = time.time()
                    blocknum +=1
            #xxx.close()
            TotalFileSize += filelength
            blocknum=0                        # File is processed, set block to 0
            FileRemnantsExists=False          # Resets the indicator of smal ending of file

            t2 = time.time()
            print round(t2-t1,2), filename

OutputTxtCSV.close()
TotalTimeEnd=time.time()
print "Total processing time=", TotalTimeEnd  - TotalTimeStart, " Seconds"

print "Finished ..."
```

### A.1.8 Python script to create block maps of hits on references to unallocated

Code A.8: CaseData_Pictogram_0.0.1.py

```
#!/usr/bin/env python
# -*- coding: utf8 -*-
# Script:   /Users/Datakrim/Qsync/PHS/UCD/UCD-2014/Pythoncode/
#           CaseData_Pictogram_0.0.1.py
# Author:   Kurt H Hansen
# Created:  2016-02-08
# Modified: 2016-02-09
# Purpose:  MSc project. Produces a pictogram of blocks in a reference
#           file with hit in unallocated clusters
# Version:  0.0.1


import sys, os, math
import encodings
import unicodedata
reload(sys)
sys.setdefaultencoding("utf8")

import mysql.connector                        # The MySQL engine
from mysql.connector import errorcode

from PIL import Image, ImageDraw

ColorRed=0
FileIdNr = (24,20,13,9,1)
BlockSize=512

def MainProcessing(cnx,cnx2, FileIdentification):
    cursor=cnx.cursor()
    SQLselect="SELECT * FROM msc_case.caselookup" + str(BlockSize) +\
                " inner join msc_case.reference_hashdatabase " +\
                " on msc_case.caselookup" + str(BlockSize) +".FileNum = msc_case.
                    reference_hashdatabase.id" +\
                " and msc_case.reference_hashdatabase.id = "  + str(FileIdentification)
    cursor.execute(SQLselect)

    # Retrieves the first file from msc-->hashdatabase
    row = cursor.fetchone()
    ReferenceBlocs = 0
    EntropyAmountRYB=[0,0,0]
    if row is not None:
        FileSize= row[8]
        FileBlocks=int(FileSize/BlockSize)
        # Set picture width to at least 600 pixels
        CreatedPictureX=max(int(math.sqrt(FileBlocks))+1,300)
        # Extra space for text in bottom of picture
        CreatedPictureY=int(FileBlocks/CreatedPictureX)+1 + 40

        print FileSize,FileBlocks, CreatedPictureX, \
                CreatedPictureY, CreatedPictureX*CreatedPictureY
        SQLselect="SELECT * FROM msc_case.caselookup" + str(BlockSize) +\
                    " where msc_case.caselookup" + str(BlockSize) +".FileNum = " + str(
                        FileIdentification) +\
                    " order by msc_case.caselookup" + str(BlockSize) +".ref_block"
        cursor2=cnx2.cursor()
        cursor2.execute(SQLselect)

        # Initialize the picture to generate

        # size of the image to create
        PictureSize = (CreatedPictureX,CreatedPictureY)
        im = Image.new('RGB', PictureSize, (192,192,192)) # create the image
        pixels = im.load()  # create the pixel map

        # Retrieves the first file from msc-->hashdatabase
        row = cursor2.fetchone()
        while row is not None:      # Continue to EOF
            ReferenceBlocs += 1
            PixelY=int(row[2]/CreatedPictureX)
            PixelX=row[2] % CreatedPictureX
            #print row[2]
            if row[3] < 0.5:
                # Entropy < 0.5 marked in black
                pixels[PixelX,PixelY] = (0, 0, 0)
                EntropyAmountRYB[2] += 1
            elif row[3] > 0.9:
                # Entropy > 0.9 marked in Red
                pixels[PixelX,PixelY] = (255, 0, 0)
                EntropyAmountRYB[0] += 1
            else:
                # Entropy 0.5 to 0.9 marked in Yellow
                pixels[PixelX,PixelY] = (255, 255, 0)
                EntropyAmountRYB[1] += 1
            row = cursor2.fetchone()
        # Create a drawing object for the picture to put some
        # text at the bottom
        draw = ImageDraw.Draw(im)
        TexColor = (0,0,0)    # color of our text in black
        text_pos1 = (5,CreatedPictureY - 36) # top-left position of our text
        text_pos2 = (5,CreatedPictureY - 24) # top-left position of our text
        text_pos3 = (5,CreatedPictureY - 12) # Bottom text
        HitPercent=round((ReferenceBlocs * 100)/FileBlocks,2)
        text1 = "B-size:"+str(BlockSize)+"   Fileid# " + str(FileIdentification) +\
                "  Hits="+str(ReferenceBlocs)+"("+str(HitPercent)+"%)"
        text2="Red=E > 0.9, Yellow=E 0.5 - 0.9 Black=E < 0.5"               # text in image
        text3="Red =" + str(EntropyAmountRYB[0])+"("+\
            str(round((EntropyAmountRYB[0]*100)/ReferenceBlocs,2)) + "%)"+\
            "  Yellow=" + str(EntropyAmountRYB[1])+"("+\
            str(round((EntropyAmountRYB[1]*100)/ReferenceBlocs,2)) + "%)"+\
            "  Black=" + str(EntropyAmountRYB[2])+"(" + \
            str(round((EntropyAmountRYB[2]*100)/ReferenceBlocs,2))+"%)"
        # Now, we'll do the drawing:
        draw.text(text_pos1, text1, fill=TexColor)
        draw.text(text_pos2, text2, fill=TexColor)
        draw.text(text_pos3, text3, fill=TexColor)
        draw.line([(0,CreatedPictureY - 40),(CreatedPictureX,CreatedPictureY - 40)], fill
            =(0,0,0), width=2)
        # Show the picture on screen
        im.show()
        # Save the picture to disk in .EPS file format
        im.save("./Pictograms/CaseData_Process_Pictogram"+ str(BlockSize)+"_"+str(
            FileIdentification) + ".eps")
    else:
        print "Database did not contain anything about that file ID"
    return True

############## Main ###################################
#
# Use the Oracle mySQL plugin to handle mySQL databases
# Open te database MSc
#
#######################################################
if __name__ == "__main__":
    for FileIdNumber in FileIdNr:
        try:
            cnx =mysql.connector.connect(user='root', host='localhost', password='root',
                database='msc_case')
            cnx2 =mysql.connector.connect(user='root', host='localhost', password='root',
                database='msc_case')
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
                print("Something is wrong with your user name or password")
            elif err.errno == errorcode.ER_BAD_DB_ERROR:
                print("Database does not exists")
            else:
                print(err)
        else:
            print("Connected to the database successfully !")
            MainProcessing(cnx,cnx2, FileIdNumber)
            cnx.close()
            cnx2.close()
    print "Finished ...... "
```

### A.1.9 Python script to create block maps of blocks located in unallocated area of pre-existing files

Code A.9: CaseData_Pictogram_Unallocated_0.0.1.py

```
#!/usr/bin/env python
# -*- coding: utf8 -*-
# Script:   /Users/Datakrim/Qsync/PHS/UCD/UCD-2014/Pythoncode/
#           CaseData_Pictogram_Unallocated_0.0.1.py
# Author:   Kurt H Hansen
# Created:  2016-02-09
# Modified:
# Purpose:  MSc project. Produces a pictogram of blocks in unallocated
#           area where a certain reference file has hits on same SHA256
# Version:  0.0.1


import sys, os, math
import encodings
import unicodedata
reload(sys)
sys.setdefaultencoding("utf8")

import mysql.connector                      # The MySQL engine
from mysql.connector import errorcode

from PIL import Image, ImageDraw

ColorRed=0
FileIdNr = (24,20,13,9,1)
BlockSize=4096
BlockResolution=100

def MainProcessing(cnx,cnx2,FileIdentification):

    row=[]
    ReferenceBlocs = 0
    if row is not None:

        SQLselect="SELECT * FROM msc_case.caselookup" + str(BlockSize) +\
                    " where msc_case.caselookup" + str(BlockSize) +".FileNum = " + str(
                        FileIdentification) +\
                    " order by msc_case.caselookup" + str(BlockSize) +".unallocated_block
                        DESC limit 1"
        cursor2=cnx2.cursor()
        cursor2.execute(SQLselect)
        row = cursor2.fetchone()
        MaxBlockNum =int(row[4]/BlockResolution)

        SQLselect="SELECT * FROM msc_case.caselookup" + str(BlockSize) +\
                    " where msc_case.caselookup" + str(BlockSize) +".FileNum = " + str(
                        FileIdentification) +\
                    " order by msc_case.caselookup" + str(BlockSize) +".unallocated_block"
        cursor2=cnx2.cursor()
        cursor2.execute(SQLselect)
        # Retrieves the first file from msc-->hashdatabase
        row = cursor2.fetchone()
        MinBlockNum = int(row[4]/BlockResolution)
        BlockRange=MaxBlockNum-MinBlockNum

        # Set picture width to at least 600 pixels
        CreatedPictureX=max(int(math.sqrt(BlockRange))+1,600)
        # Extra space for text in bottom of picture
        CreatedPictureY=int(BlockRange/CreatedPictureX)+1 + 30

        print MinBlockNum, MaxBlockNum, CreatedPictureX, CreatedPictureY
        #

        # Initialize the picture to generate

        # size of the image to create
        PictureSize = (CreatedPictureX+2,CreatedPictureY)
        im = Image.new('RGB', PictureSize, (255,255,255)) # create the image
        pixels = im.load()  # create the pixel map

        while row is not None:      # Continue to EOF
            ReferenceBlocs += 1
            row4_value=int(row[4]/BlockResolution)
            PixelY=int((row4_value-MinBlockNum)/CreatedPictureX) + 1
            PixelX=( (row4_value-MinBlockNum) % CreatedPictureX )+ 1
            pixels[PixelX,PixelY] = (255, 0, 0)
            row = cursor2.fetchone()

        # Create a drawing object for the picture to put some
        # text at the bottom
        draw = ImageDraw.Draw(im)
        TexColor = (0,0,0)    # color of our text in black
        text_pos1 = (5,CreatedPictureY - 24) # top-left position of our text
        text_pos2 = (5,CreatedPictureY - 12) # top-left position of our text

        text1 = "B-size:"+str(BlockSize)+"   Fileid# " + str(FileIdentification) +\
                "  Hits="+str(ReferenceBlocs)
        text2="Red=Block ref   Res: "+str(BlockResolution)+" blocks   "+\
                "Range u.c: " +  str(MinBlockNum*BlockResolution)+"-"+str(MaxBlockNum*
                    BlockResolution) +\
                " Range u.b.512: " + str(MinBlockNum*BlockResolution*int(BlockSize/512))+"-
                    "+str(MaxBlockNum*BlockResolution*int(BlockSize/512))   # text in image
        # Now, we'll do the drawing:
        draw.text(text_pos1, text1, fill=TexColor)
        draw.text(text_pos2, text2, fill=TexColor)
        Graphic_DrawLines(draw, CreatedPictureX, CreatedPictureY)

        im.show()
        # Save the picture to disk in .EPS file format
        im.save("./Pictograms/CaseData_Process_Pictogram_Unallocated"+ str(BlockSize)+"_"+
            str(FileIdentification) + ".eps")
    else:
        print "Database did not contain anything about that file ID"
    return True


def Graphic_DrawLines(draw, maxX,MaxY):
    draw.line([(0,MaxY - 27),(maxX+1,MaxY - 27)], fill=(0,0,0), width=1)
    draw.line( [(0,0), (maxX+1,0) ], fill="black", width=1)
    draw.line( [(maxX+1,0), (maxX+1,MaxY-1) ], fill="black", width=1)
    draw.line( [(0,MaxY-1), (maxX+1,MaxY-1) ], fill="black", width=1)
    draw.line( [(0,0), (0,MaxY-1) ], fill="black")

############## Main ###################################
#
# Use the Oracle mySQL plugin to handle mySQL databases
# Open te database MSc
#
#######################################################
if __name__ == "__main__":

    for FileIdNumber in FileIdNr:
        try:
            cnx =mysql.connector.connect(user='root', host='localhost', password='root',
                database='msc_case')
            cnx2 =mysql.connector.connect(user='root', host='localhost', password='root',
                database='msc_case')
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
                print("Something is wrong with your user name or password")
            elif err.errno == errorcode.ER_BAD_DB_ERROR:
                print("Database does not exists")
            else:
                print(err)
        else:
            print("Connected to the database successfully !")
            MainProcessing(cnx,cnx2,FileIdNumber)
            cnx.close()
            cnx2.close()
    print "Finished ...... "
```

### A.1.10 Python script to calculate average entropy on file types

Code A.10: Entropy_Calculate_Files_0.0.1.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:   /Users/Datakrim/Dropbox/PHS/UCD/UCD-2014/Pythoncode/
#           Entropy_Calculate_Files_0.0.1.py
# Author:   Kurt H Hansen
# Created:  2016-02-16
# Modified:
# Purpose:  MSc project
#           Find average entropy on a type of file
#           (like pdf files or docx files)
# Version:  0.0.1


import sys, os, hashlib, datetime,entropy,time

reload(sys)
sys.setdefaultencoding("latin-1")
t_start=0
count_Entropy=0
count_Files=0
# t1=str(datetime.datetime.now())
with open("/Volumes/ExFAT_1TB/Entropy_DOCX.txt", "w") as outputFile:
    for path, subdirs, files in os.walk(r'/Volumes/ExFAT_1TB/DOCX/'):
        for filename in files:
            f = os.path.join(path, filename)
            try:
                f_size= os.path.getsize(f)
            except:
                f_size=0
            if f_size > 0 and filename[0] != '.' and filename[0] != '_':
                try:
                    t_start=time.time()

                    FileContent=open(f).read()
                    Entropy = entropy.shannon_entropy(FileContent)

                    #print f, Entropy

                    outputFile.write(str(f)+";"+str(f_size)+";"+str(Entropy) + os.linesep)
                    t_end=time.time()
                    print str(round(t_end-t_start,4)), Entropy, str(f)
                    count_Files +=1
                    count_Entropy += Entropy
                except:
                    pass
                    #print "Error in reading file:", f
print "Average entropy on " + str(count_Files)+" files is: "+str(count_Entropy/count_Files)
```

### A.1.11 Python script to extract unallocated blocks from NTFS file system

Code A.11: NTFS_ReadUnallocatedFromBitmap_0.0.1.py

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# Script:
```

```
# Author:   Kurt H Hansen
# Created:
# Modified:
# Purpose:
# Version:  0.0.1


import sys, os
import struct
import encodings
import unicodedata
reload(sys)
sys.setdefaultencoding("latin-1")
global clusterSize
global f_bitmap, f_volume, f_unallocated
clusterSize= 4096


def BitmapCheckStatus(status,bitmapOffset):
    offsetInVolume=clusterSize * bitmapOffset
    for bit in range(0,8):
        if status & (2 ** bit) <= 0:
            f_volume.seek(((bitmapOffset * 8)+bit) * clusterSize,0)

            #print bitmapOffset, clusterSize, bit, ((bitmapOffset * 8)+bit) * clusterSize

            Block=f_volume.read(clusterSize)
            f_unallocated.write(Block)
    return True


f_bitmap=open('/Volumes/ExFAT_1TB/msc_case_image/Bitmap', 'rb')
f_volume=open('/Volumes/ExFAT_1TB/msc_case_image/8916796-2013_1732_A5.dd', 'rb')
f_unallocated=open('/Volumes/ExFAT_1TB/msc_case_image/Unallocated1', 'wb')

bitmapSize=os.path.getsize('/Volumes/ExFAT_1TB/msc_case_image/Bitmap')

for i in range(0, bitmapSize):
    f_bitmap.seek(i)
    x=f_bitmap.read(1)

    bitmapStatus = struct.unpack('<B', x)

    BitmapCheckStatus(bitmapStatus[0],i)

f_bitmap.close()
f_volume.close()
f_unallocated.close()
print "Finished ..."
```

## A.2 SQL Queries

Code A.12: Inner join of all records in case vs references

```
LOAD DATA INFILE 'path/filename.csv' INTO TABLE Blockhash512 FIELDS TERMINATED BY '\t'
    LINES TERMINATED BY '\n' IGNORE 0 ROWS (FileNum,SHA256,BlockNum,Entropy);
```

Code A.13: SQL command to detect collision in tables

```
select SHA256,count(*) as count from blockhashNNNN group by sha256 having count(*) > 1;
```

Code A.14: SQL command to detect mutual blocks in reference data and unallocated

```
use msc_case;
select reference_blockhash4096.FileNum, reference_blockhash4096.SHA256,
    reference_blockhash4096.BlockNum as Reference_Blocknum, reference_blockhash4096.Entropy
    , blockhash_unallocated4096.BlockNum as Unallocatd_Blocknum  from
    blockhash_unallocated4096,reference_blockhash4096  where blockhash_unallocated4096.
    SHA256 = reference_blockhash4096.SHA256  and reference_blockhash4096.FileNum > 0  and
    reference_blockhash4096.Entropy > 0.01  order by reference_blockhash4096.FileNum,
    reference_blockhash4096.BlockNum  INTO OUTFILE 'r:\
    LookupReferenceToBlockhash_size4096_FileID_ALL.csv' FIELDS TERMINATED BY ',' LINES
    TERMINATED BY '\n';
```

Code A.15: SQL command to detect collision in sub-query

```
use msc_case;
SELECT count(FileNum) as Records, FileNum FROM msc_case.caselookup512 group by FileNum
    order by Records;
```

Code A.16: SQL command to perform Injection-Verification

```
SELECT reference_blockhash4096.FileNum AS "Reference file",
   reference_blockhash4096.BlockNum AS "Reference block",
   reference_blockhash4096.SHA256,
   Blockhash_unallocated4096.BlockNum AS "Unallocated block"
FROM reference_blockhash4096
INNER JOIN Blockhash_unallocated4096
ON reference_blockhash4096.SHA256 = Blockhash_unallocated4096.SHA256
WHERE reference_blockhash4096.FileNum = 1
ORDER BY Blockhash_unallocated4096.BlockNum;
```