Utilising Reduced File Representations to Facilitate Fast Contraband Detection

Sean McKeown

School of Computing
Edinburgh Napier University

A thesis submitted in partial fulfilment of the requirements of Edinburgh Napier University, for the award of
Doctor of Philosophy.

October 2019

This work is dedicated to my parents, Joe and Margo, who instilled the value of education in me at an early age, and to Colette for her loving support and inexhaustible supply of belief in me.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Sean McKeown
October 2019

Acknowledgements

First of all I would like to thank Dr Peter KK Lee who provided the funding for my PhD studentship, making this entire journey possible.

Additionally I would like to express my gratitude to Dr Leif Azzopardi and Dr Brad Glisson for their support in the early years of my research, which helped to prepare me for my PhD at Napier and allowed me to hit the ground running. The PhD students and early career researchers at Glasgow also helped enormously and I'd particularly like to thank Dr David Maxwell and Dr Yashar Moshfeghi.

I have of course received a lot of support from my supervisors at Napier: Dr Gordon Russell, Dr Petra Leimich, and Prof. Bill Buchanan, who have made it possible for me to build a respectable publication track record during my time as a research student. The support and encouragement I received was invaluable. I'd also like to acknowledge the PhD student community at Napier who made the experience so much better.

Finally, I appreciate the time given by Dr Colette Bowie in proof reading and providing general linguistic support for my thesis and publications - it was a great help.

Abstract

Digital forensics practitioners can be tasked with analysing digital data, in all its forms, for legal proceedings. In law enforcement, this largely involves searching for contraband media, such as illegal images and videos, on a wide array of electronic devices. Unfortunately, law enforcement agencies are often under-resourced and under-staffed, while the volume of digital evidence, and number of investigations, continues to rise each year, contributing to large investigative backlogs.

A primary bottleneck in forensic processing can be the speed at which data is acquired from a disk or network, which can be mitigated with data reduction techniques. The data reduction approach in this thesis uses reduced representations for individual images which can be used in lieu of cryptographic hashes for the automatic detection of illegal media. These approaches can facilitate reduced forensic processing times, faster investigation turnaround, and a reduction in the investigative backlog.

Reduced file representations are achieved in two ways. The first approach is to generate signatures from partial files, where highly discriminative features are analysed, while reading as little of the file as possible. Such signatures can be generated using either header features of a particular file format, or by reading logical data blocks. This works best when reading from the end of the file. These sub-file signatures are particularly effective on solid state drives and networked drives, reducing processing times by up to 70$\times$ compared to full file cryptographic hashing. Overall the thesis shows that these signatures are highly discriminative, or unique, at the million image scale, and are thus suitable for the forensic context. This approach is effectively a starting point for developing forensics techniques which leverage the performance characteristics of non-mechanical media, allowing for evidence on flash based devices to be processed more efficiently.

The second approach makes use of thumbnails, particularly those stored in the Windows thumbnail cache database. A method was developed which allows for image previews for an entire computer to be parsed in less than 20 seconds using cryptographic hashes, effecting rapid triage. The use of perceptual hashing allows for variations between operating systems to be accounted for, while also allowing for small image modifications to be captured in an analysis. This approach is not computationally expensive but has the potential to flag illegal media in seconds, rather than an hour in traditional triage, making a good starting point for investigations of illegal media.

Table of contents

List of Figures

List of Tables

List of Publications

Portions of the work presented in this thesis are included in the following peer-reviewed publications, listed in chronological order:

1. McKeown S, Russell G, Leimich P. Fast Filtering of Known PNG Files Using Early File Features. In: Annual ADFSL Conference on Digital Forensics, Security and Law. Daytona Beach, Florida, USA: ADFSL; 2017. Available from: https://commons.erau.edu/adfsl/2017/papers/1/

2. McKeown S, Russell G, Leimich P. Fingerprinting JPEGs With Optimised Huffman Tables. Journal of Digital Forensics, Security and Law. 2018;

3. McKeown S, Russell G, Leimich P. Sub-file Hashing Strategies for Fast Contraband Detection. In: IEEE International Conference on Cyber Security and Protection of Digital Services (Cyber Security 2018). Glasgow, UK: IEEE; 2018.

4. McKeown S, Russell G, Leimich P. Reducing the Impact of Network Bottlenecks on Remote Contraband Detection. In: IEEE International Conference on Cyber Security and Protection of Digital Services (Cyber Security 2018). Glasgow, UK: IEEE; 2018.

Chapter 1

Introduction

The modern world is abundantly interconnected, with approximately half the population of the planet making use of the Internet [5]. Technological advancements have placed high resolution cameras and Internet connected mobile devices in the hands of billions of people, which, together with the rise of social media, has positioned many consumers as data producers, as well as data consumers. Indeed, 90% of all data in existence was created in the last two years, at approximately 2.5 exabytes per day [6]. By 2020, it is expected that on average 1.7MB of data will be produced every second for every person on earth.

In a digital world, some data inevitably becomes associated with criminal activity, either due to criminals directly exploiting new technologies, or as background evidence in a non-computerised crime. Digital evidence requires expert analysis in order to interpret it and avoid manipulation or unintentional destruction of evidential artefacts. For this reason the field of digital forensics exists, with both private and public sector institutions employing such trained individuals. However, the explosive growth in the quantity of data generated and stored by the average person, together with an increasing number of investigations involving digital evidence, has placed digital investigators at a disadvantage. As a result, the 'Golden Age' of digital forensics has ended [7], as existing tools, software, and processes have failed to adapt at an appropriate rate. This has left the forensics community in a difficult position, necessitating an ongoing search for solutions [7].

1.1 Motivation

Forensics approaches which were designed for hard disks with capacities on the order of several gigabytes have not transitioned well to terabyte scale devices, nor to the increasingly heterogeneous digital environment. The so called volume problem in digital forensics has been an issue for some time [8], stemming originally from dramatic increases

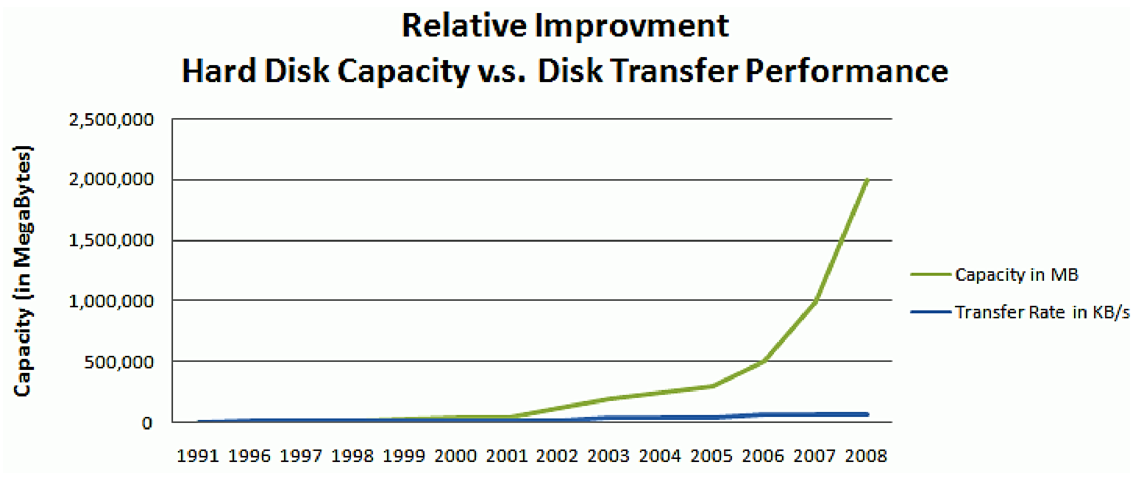


Fig. 1.1 The relative increases in disk capacity versus disk read speeds between 1991 and 2008. Sourced from http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608

in hard disk capacities in the late ’90s, where disk density doubled each year, gradually slowing to 20–30% per annum increases in the modern day [9]. The volume problem is compounded by the failure of disk read speeds to keep pace with the increases in capacity, depicted in Figure 1.1. As the gap between read performance and capacity has increased, the time taken to read all data on a disk has also increased. To illustrate, a typical hard disk in 1991 could read its entire contents in a few minutes[1] [10], increasing to an hour in 2003[2] [11], while a modern 3TB hard drive takes approximately 8 hours[3] [12]. In practice, these maximal read speeds are not sustained when producing forensic copies of a device, which can take over 11 hours for a 3TB drive [11], prior to any actual analysis being started. While the acquisition time may be substantially lower on Solid State Drives (SSDs), it is still relatively expensive. Similar acquisition bottlenecks are also found in networked environments. Network bottlenecks will likely play an increasingly large role in many digital investigations, as more criminal activity transitions to cloud environments [13]. Such Input/Output (I/O) and bandwidth constraints mean that in many scenarios accessing data is more expensive than processing it, such that systems must be engineered with this I/O bottleneck in mind.

Forensic investigators are being inundated with data, not only from the impact of larger storage capacities, but also due to the increases in the number of cases, and in the number of devices per case [14, 15]. Unfortunately, many law enforcement departments are short on resources and are unable to keep up with demand [16], resulting in public sector forensics backlogs of up to 18 months in the UK (2015) [17], and four years in Ireland

[1]Maxtor 7040a, benchmarked at 600kB/s, takes 66.66 seconds for the 40MB version and 3.33 minutes for the 140MB version.

[2]200GB at 58MB/s, drive model absent in cited work [11].

[3]7.9 hours, 116MB/s read speed, benchmarked from a 3TB WD RED (2012).

(2016) [18]. This large lag time before digital evidence is examined creates a number of problems. When the subject is guilty, police interviewers are placed at a disadvantage as they lack information, while the same information may be used to prevent ongoing crimes or locate existing victims [19]. This delay also makes it difficult to seek access to the logs of Internet services, which are typically only retained by service providers for several months, resulting in lost evidence. When the subject is innocent, there is a tangible loss of equipment and access to personal data during the investigation, while social pressures can destroy lives, occasionally resulting in suicide [20]. Additionally, defence lawyers may not have timely access to forensics reports which are required to mount a sound legal defence [19], unfairly prejudicing the innocent.

To begin to address the backlog problem in digital forensics, new approaches are required. Such approaches must be substantially faster while maintaining accuracy and should allow for investigations to be closed far more rapidly, both to account for future increases in data, but also to begin to chip away at the years of backlogs. The efficacy and applicability of digital forensic investigations will be shaped by the effectiveness of the data triage and data reduction tools which are being built today.

1.2 Thesis statement

This thesis states that reduced representations of files may be used to mitigate the I/O and bandwidth bottlenecks present in current digital investigations, leading to faster forensic processing. Stand-in file representations, such as partial files or thumbnails, may be used to reduce the volume of data to read from storage media, while still maintaining forensic integrity and accuracy. Ultimately, with the current state of the digital forensics landscape, some trade-offs may have to be made between speed and completeness if any headway is to be made in tackling the volume problem.

This work focuses on a subset of forensics processing relating to the identification of known contraband files, which may relate to cases involving child abuse. As the detection of this material is prominent in the majority of public sector investigations, decreasing the time taken to carry out this type of task will ideally improve case turnaround time and begin to address backlogs.

1.3 Contributions and Origins of the Material

This thesis has three primary contributions:

1. The development of file type specific, partial file, signature generation techniques. These approaches exploit properties of a given file type to produce discriminative

signatures which can be used to rapidly detect contraband on a device. Such signatures typically require reading 1–3%[4] of a relatively small file, resulting in substantial data reduction. Benchmarks show this approach to be particularly good on solid state media, while also achieving substantial performance gains on hard drives with images of reasonable size. Proofs of concept were developed for the PNG and JPEG formats, which have been published in the ADFSL conference [1] and journal [2], respectively.

2. A generic approach to sub-file signature generation which does not rely on file type specific features. This approach makes use of small block data hashes from various parts of the file, requiring as little as 4KiB of data per image to produce unique signatures at the million image scale. The amount of data to read per file is fixed, with an associated fixed length acquisition time, regardless of actual file size. This approach has similar performance characteristics to the file type specific approach, above, reducing processing times significantly. This contribution was published as two separate papers at the IEEE Cyber Security conference: the first paper deals with local disk benchmarks [3], while the second performs benchmarks on networked storage devices [4].

3. A very fast thumbnail based triage technique for Windows, which has the potential to assess a drive of any size in a matter of seconds. This approach is also potentially extensible to other operating systems and may be used prior to processing the remainder of the disk with the sub-file approaches identified above. Thumbnail analysis was also conducted for cloud based services, with the reduced data requirements facilitating faster contraband detection on cloud storage. The Windows based triage approach has been submitted for publication in the Journal of Digital Forensics, Security and Law.

The initial inspiration for this thesis, particularly the material in Contribution 1, came from the work of Edmundson and Schaefer [21–24] in the field of compressed domain Content Based Image Retrieval (CBIR), in which it was determined that JPEG compression tables can be used to communicate information about the content of an image. The authors use JPEG encoding tables to act as a proxy for the full image when conducting a Reverse Image Search, finding it to be incredibly fast and surprisingly effective. Elements of header based forensics [25–27], where metadata is used to verify the source of an image, were also incorporated to fuse knowledge from the two distinct disciplines. The work for Contribution 1 extends these ideas by using such metadata structures to identify a particular

[4]This percentage can occasionally be significantly inflated by large quantities of metadata, such as Adobe Photoshop XMP and embedded thumbnails.

image, rather than perform similarity searches or identify source cameras/software. The technique is also extended to the PNG format to explore how it generalises to other file formats. In order to make the approach agnostic to file types, an alternative method was explored which focused on cryptographic hashing strategies for logical file subsets. This effectively extended the work on sub-file hashing in forensics [28, 29] which previously focused only on breaking files into blocks at physical disk sector boundaries. The latter parts of the thesis for Contribution 3 draw more generally on the idea of using data reduction methods in forensics [30], while making reference to the existing work in thumbnail forensics [31, 32]. The main expansion for this contribution was to explore the possibility of using a single hash database which can be used to detect contraband in centralised thumbnail caches across a variety of operating system releases/versions, which is difficult due to the lack of standardised thumbnailing approaches. The use of Perceptual Hashing was explored for use in this context, a set of techniques which have not seen a great deal of application thus far in digital forensics.

By synthesising knowledge and approaches from a variety of research areas, the work in this thesis demonstrates that reduced file representations are an effective tool for mitigating I/O bottlenecks in forensics. Substantial speed improvements in detecting contraband will allow both forensic triage and deep analyses to be executed more swiftly, allowing the forensic practitioner to spend more time answering investigative hypotheses than was previously possible in the time-constrained world of public sector forensics. Additionally, in showing that such reduced representations are robust enough for the forensic use case, this thesis hopes to stimulate the development of new approaches to forensics which deterministically leverage partial-files for contraband detection. The time saved by such techniques will ideally reverse the trend of law enforcement backlogs and allow the criminal justice system to function as it was intended.

It should be noted that physical read speed limitations are not the only bottleneck in current digital forensics processes. Other aspects of an investigation, such as device seizure, device/case prioritisation and device/case level analyses all contribute to the overall turnaround time of an investigation. In complicated cases the analysis phase of the investigation can take up to 2 months [16]. The approach in this work does not necessarily reduce the time taken to perform extensive device analyses, however by focusing on fast processing in the context of triage, it may be possible to seize fewer devices, or better prioritise evidence items, which will have an effect on the time taken to conduct the entire investigation. In cases where there are few devices and complex forensic analysis is involved, perhaps in the case of a sophisticated user deploying anti-forensics, speeding up the raw processing time may not be particularly helpful, as this is not the main challenge in such an investigation. Indeed, in cases where files are encrypted, heavily modified or otherwise obfuscated, most signature based approaches will be of little use. The work in this thesis, then, focuses on the case where rapid processing of contraband items can be used to quickly direct an investigation or facilitate rapid triage, but may not be of substantial benefit to every form of forensic investigation

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 provides an overview of relevant literature. It begins with a discussion of digital forensic triage, which aims to reduce the turnaround time for an investigation and deal with forensic backlogs. Signature generation techniques are then explored, which are particularly relevant for contributions 1 and 2. Finally, literature on how reduced file representations have been used in a variety of fields is examined.

Chapter 3 first outlines the requirements for an effective file signature, before discussing sub-file signature approaches for the PNG and JPEG formats, corresponding to contribution 1. The latter part of the chapter deals with practical concerns, such as how these techniques may behave on future storage media, and how they could fit into existing digital forensics processes.

Chapter 4 generalises sub-file approaches with the introduction of the file type agnostic hashing strategies of contribution 2. This work builds on Chapter 3, with comparisons between the two approaches included towards the end of the chapter. These sub-file techniques are examined for both local disk and networked storage environments in order to better understand the performance properties of the proposed algorithms.

Chapter 5 deviates from the sub-file approach, instead relying on existing thumbnails which are potentially present in an operating system's thumbnail cache, as well as on cloud storage services. The bulk of this work focuses on rapid disk level triage for Windows 10, but is extensible to other operating systems. A small case study on Dropbox demonstrates the applicability of the technique on cloud storage systems, completing the analysis for contribution 3.

Chapter 6 provides a discussion of this work and derives overall conclusions for the thesis. This is then followed by suggestions for future work.

Chapter 2

Background and Literature Review

2.1 Introduction

With the challenges of dealing with large digital forensic datasets outlined, this chapter explores the problem in more depth with reference to relevant literature. Section 2.2 focuses on forensics approaches which attempt to tackle the current data volume crisis, before discussing current methods of generating contraband file signatures in Section 2.3. Finally, Section 2.4 examines how existing reduced file representations are created, and exploited, across several computing disciplines.

2.2 Digital Forensic Triage

2.2.1 Coping with Volume

The huge volume of data present in modern digital forensics investigations has necessitated the formulation of coping strategies to prevent digital forensics practitioners from being overwhelmed. Digital Forensic Triage is the collective term for such a class of strategies, which can be defined as a way of maximising effective resources by reducing the amount of data which needs to be examined in full [33]. This can be achieved via administrative triage, where the seriousness of the crime and other contextual factors are used to decide if a case is re-prioritised or dropped entirely. The alternative, technical triage, seeks to provide rapid insight into the potential evidential utility of a particular device, such that it can be seized for additional processing if appropriate, or omitted from deeper analysis based on the findings of the triage process. Technical triage typically makes use of device previews to quickly ascertain if a device has ‘low hanging fruit’ [34] evidence items. While technical triage has drawbacks, in that a critical piece of evidence may be missed, there is a preference among police detectives, with no forensic expertise, for rapid delivery of partial evidence, rather than waiting long periods for complete reports [33]. An additional benefit is that decisions are based on facts derived directly from suspect devices, rather than being motivated by external factors.

The necessity of triage is indicative of a failure of forensic tools and processes to adapt to the modern digital landscape [33], such that additional research is required to prevent backlogs from growing further. Indeed, without intervention, backlogs are likely to increase at a faster rate as the volume of investigative data is predicted to balloon further in the future [18]. A 2014 review of the literature by Quick and Choo [15] suggested that while some progress had been made, there is still much scope for improvement, with the most promising areas of research for dealing with backlogs being data reduction, data mining and intelligence analysis. A more recent survey of international practitioners dealing with indecent images of children [16] indicated that any subsequent advances have not helped stem the tide, with reports that processing times are still very long, with some analyses taking up to 2 months. The resources provided to practitioners have simply not scaled with the increase in workload [35], as the number of images and videos per device, and the number of cases, continues to swell [15]. Indeed, the lack of sufficient numbers of digital forensics specialists has resulted in the development of triage models which utilise non-specialist first responders [36], reducing the workload of forensics experts and allowing them to focus on deeper forensic analyses which better leverage their expertise.

Roussev et al. [11] suggest that forensics tools should prioritise performance as highly as reliability and correctness, with a particular focus on the time lapse, or latency, taken to acquire initial results. Answering the investigative questions, "who, what, when, where, and how" [37] as early as possible is important and will shape the remainder of the investigation and analysis.

Done correctly, technical triage can begin to fill important knowledge gaps about a device or investigation. However, Shaw and Browne [34] note that many triage approaches do not effectively reduce the amount of data to process, as devices which contain evidence are ultimately still fully analysed from scratch after initial triage. The time taken on this subsequent full analysis has associated risks, such as innocent subjects committing suicide, or guilty parties being conferred reduced sentences due to time waited. In order to truly begin to address the backlog, Shaw and Browne [34] suggest that evidence found during triage should give the forensic practitioner a head start on the more complete analysis.

2.2.2 Triage Solutions

This section discusses a variety of existing approaches to evidence based, technical, triage. The goal is typically to provide some usable forensic indicators as fast as possible, though some techniques also facilitate faster deep analyses of a device.

Shaw and Browne [34] describe the ‘enhanced previewing’ approach to forensic triage, where a Compact Disc (CD) is used to boot a suspect device for analysis, with preview items being stored on an external drive. Previewed data includes images, storyboarded videos, text string searches, registry and chat log parsing, and other artefacts, with some configuration for investigation type. The risk of missing information is mitigated by processing all of the device, however, it is not clear how this can be done in a timely fashion on large modern devices, particularly when the point of doing triage in this way is to avoid creating unnecessary forensics copies of benign devices.

Roussev and Quates [38] assert that a primary bottleneck in the forensic analysis of a device is the time it takes to read data from the device, which will be governed by the read speed, or throughput, of the storage media. The authors propose an approach where a full forensic copy of a device is produced at the same time as data is processed for triage, using a heavily-parallel processing model. This approach essentially addresses the need to have immediate results, while removing unproductive downtime waiting for forensic copies for the full analysis stage. The authors present a case study using the similarity hashing tool *sdhash* to correlate evidence across multiple devices. Doing this in parallel on a powerful 24-core workstation allowed for overviews of various scenarios to be sketched out in under 2 hours.

The *bulk_extractor* tool, described by Garfinkel [39] also uses a stream based approach and processes data during forensic copying. This is achieved by implementing multiple scanners, which process images, documents, and other textual data to extract artefacts such as email addresses, telephone numbers, and credit card numbers. Generated histograms provide a visual overview of potentially important information, such as the most frequently occurring email addresses.

Taking the above parallel processing methods to their logical conclusion, Roussev et al. [11] expand the approach to a much wider range of analyses which would be conducted as part of a full investigation of a device. Individual analyses are handled by a variety of worker nodes, calculating traditional cryptographic and similarity hashes, parsing windows registry and metadata information, indexing text, and decompressing files. Again, processing is conducted with the intention of keeping up with the rate at which data is read sequentially from disk. Unfortunately, the authors note that only traditional cryptographic hashing was computationally inexpensive enough to achieve this goal in real time on a 48-core server. However, approaches which use small amounts of data, such as registry analysis, are fast enough in practice. The authors suggest that 120–200 cores are required to keep pace with a data stream of 120MB/s, while still requiring the full capture duration to read every bit of data from the disk to complete processing. Performing more expensive processing, such as perceptual hashing [40] or steganography analysis would further inflate the CPU requirements of such a strategy, likely making it infeasible. Additionally, despite recent advances in processor technology, it will be unlikely to cope with the typical 500MB/s sustained stream produced by a SATA3 SSD. Given that this approach requires expensive equipment from already under resourced forensics departments [16, 35], this approach may not be the most practical solution.

The rate of data streaming is a bottleneck, but it is also apparent that processing power can be a bottleneck if complex analyses are attempted to be carried out in real time. One method of coping with both of these challenges is to apply data reduction techniques to the content of a particular device. Reading less data from a device cuts down on the impact of the read speed bottleneck, while potentially reducing the complexity of later analyses. Quick and Choo [30] propose an additional step in standard forensics processes wherein this data subsetting is applied to either forensic images or write blocked source devices. A logical image is created from the subset, which contains a variety of information such as Internet history, keyword searches, Windows registry artefacts and hash database lookups. The resulting subset is roughly 0.2% of the size of the original media and can be indexed and processed much faster than a full device. A case study showed that this approach allowed for the subset to be acquired from a 320GB drive in 72 seconds, while a full forensic copy took three hours. A later expansion of this work [41] defines more clearly which data to acquire, with the addition of items such as file system metadata, event logs, thumbnails, chat logs, documents, media player data and other critically relevant information. This expanded collection took a median of 14 minutes to acquire and process in Internet Evidence Finder tool [42], which is considerably faster than the eight hours required to produce a full forensic image for the same dataset.

A similar idea is pursued by Grier and Richard [43] wherein investigation-specific filters are used to select data subsets to acquire. The primary difference here is that data subsets are physical bit-level images of disk regions. Experiments yield a collection acceleration of over 3× while still obtaining 95% of the evidence. Both Grier and Richard [43] and Quick and Choo [30, 41] potentially miss information based on their collection heuristics and case-specific criteria, but what is collected could be re-processed at a later time with more comprehensive tools. One weakness of these approaches is that they make assumptions about the content of a particular device, relying heavily on there being an operating system installed on it. This filtered subsetting approach would not reduce the time to analyse or collect data from secondary drives which are only used to store media, such as those used by a NAS.

The above data reduction approaches are deterministic, in that the same subsets will always be generated from a given device using a particular configuration. They also either attempt to process an entire disk in some fashion or use a set of heuristics to limit which data is captured. Young et al. [29] take an alternative approach, reducing the amount of data to read from a disk by means of random sampling. A sample of disk sectors is chosen using a pseudorandom number generator, with each sector being cryptographically hashed and compared to a database of equivalently sized block hashes for known files. In doing so, technical triage becomes a statistically quantifiable endeavour, with known false negative rates. Additionally, this approach is completely independent of the file system, with the only requirement being that there is some alignment between sampled disk blocks, file data blocks, and disk sectors. This alignment depends on the sample unit size and the native disk sector size which was traditionally 512 bytes, but more recently 4096 bytes (4KiB) on modern drives.

Breaking a file into blocks and using block hashes for detection purposes may result in a number of false positives, as not all file blocks are unique to a particular file. Common, ‘non-probative’ blocks are discussed by Garfinkel and McCarrin [44], where a number of heuristics are devised to identify such common blocks. An additional problem is the memory space required to store these block hashes. When a single hash is used for the entire file, traditional algorithms such as MD5 produce a fixed length digest of 128 bits, regardless of the file size. When hashing blocks, the number of 128 bit hashes needed is multiplied by the number of blocks in the file. For example, a 40KiB file would require 1280 bits to store all block hashes, in contrast to a single full file hash of just 128 bits. As this approach is most useful for deciding whether or not a device should be seized, it should be able to run on a wide variety of low specification systems. Penrose et al. [45] reduce the memory overheads required by making use of Bloom filters to store the hash database, allowing it to fit in the memory of legacy equipment. The cost is a small increase in the number of false positives caused by the probabilistic nature of Bloom filters. However, this is a quantifiable design choice and does not increase the chance of false negatives. Benchmarks of this approach [45] show that the processing time depends on sample size and computer specifications, ranging from seconds on modern machines with Solid State Drives, to less than an hour in most cases with modern hard disks. By selecting an appropriate number of samples, as little as 4MiB of data of interest can be detected with very high probability on a large disk in a short time. More recent work [46] has sought to further enhance block based sampling by breaking the disk into regions and sampling consecutive blocks within each region. As regions are relatively small this minimises disk read head seek times and reduces the total number of read operations required to sample a device. However, this may reduce the total number of effective samples, and must be controlled for when selecting the total number of samples/regions.

While random sampling approaches have been shown to be effective for device triage, they only offer a minimal starting point for the full analysis which occurs afterwards, and thus do not necessarily reduce the time taken to analyse devices which are selected from the triage stage.

The work described above makes it clear that there are physical and computational limitations to current forensics methods. Disk read speeds are often a limiting factor, however, a full analysis cannot be feasibly conducted in real time even at these limited speeds. Data reductions strategies appear to be the most promising approach, with several techniques allowing for reduced acquisition and processing times, at the cost of some uncertainty. Random sampling is a promising technique for initial triage, but the results are not easily fed into the full examination stage, and therefore would not necessarily help in reducing backlogs caused by long device analysis times. The reduced file representations in this thesis fall into the data reduction category, with the primary focus being on contraband detection. It would therefore be instructive to review existing methods for generating contraband signatures for automatic detection, which is presented in the following section.

2.3 Signature Generation Techniques for Contraband Detection

Whether part of forensic triage or the full analysis stage, data signatures are integral to the digital forensics process, being used to whitelist or blacklist previously encountered files and to verify that the integrity of digital evidence has not been compromised. One important use case for this kind of signature is the automatic detection of contraband images and videos, such as those relating to child exploitation, which makes use of large hash databases [47, 48] to detect previously encountered media. This particular investigation type constitutes the majority of public sector investigations [13].

File signatures are typically created by calculating cryptographically secure hash digests, producing binary signatures which are both distinct and extremely unlikely to be shared by two input files. The traditional approach is to hash every bit of data in each input file using algorithms such as MD5 [49] and the SHA family of algorithms [50]. However, as noted by Kornblum [51], traditional cryptographic hashes can easily be foiled by modifying even a single bit of the input data, resulting in a completely different hash digest. This means that criminals can make trivial modifications to avoid this kind of detection and that very similar or re-encoded files are also not detectable without being hashed previously in a lookup database.

In order to provide a level of tolerance for detecting slightly modified file variants, or largely similar files, different approaches are required. Signature generation techniques that allow for the detection of similar files are broadly labelled as *approximate matching* methods. Breitinger et al. [52] classify approximate matching methods into three categories:

Cryptographic Block Hashing: Where traditional hashes such as MD5 and SHA1 are applied to the binary data of a file which is broken into blocks, with similarity scores being based on the number of overlapping blocks [53]. This method fails completely if a single bit in each block is modified, but is inexpensive to employ and can detect very small fragments of contraband files.

Bytewise Approximate Matching: Bytewise methods also work at the binary level but are more robust to small manipulations and sector alignment issues. These approaches are suitable for detecting binary data variants and file fragments. They are more computationally complex than traditional hashes and are limited to detecting binary level similarities between files.

Semantic Approximate Matching: Semantic methods do not work at the binary level, instead they derive signatures from the human-facing content of each file. For images and videos this is equivalent to signatures based on the approximate visual properties of the media. Using this approach it is possible to detect similar file content even when the underlying binary data representation is significantly different.

The following subsections discuss the latter two approximate matching approaches in more detail as they inform decisions made in this thesis when creating reduced file representations.

2.3.1 Bytewise Approximate Matching

The weaknesses of traditional hashing led Kornblum [51] to propose Context Triggered Piecewise Hashes (CTPH), adapting prior work in spam detection for forensics purposes. A rolling hash is calculated from a sliding window across binary data, generating hashes at trigger points based on the input. These trigger points essentially split the file into chunks, which are then independently hashed using the FNV algorithm [54]. The six least significant bits from each chunk hash are then concatenated to produce the final hash. The similarity of two hashes is given by the weighted edit distance[5], before being rescaled and inverted. A score of zero represents 'no homology', while a score of '100 indicates almost identical files'. The tool *ssdeep* [51], allows for similar files to be detected while being resistant to small changes. Modifying a single bit in the input file represents a small change in the overall hash, resulting in a high match score being generated. One key factor which separates this approach from comparing block based hashes is that the triggers for chunk hashes ignore disk block alignment, meaning that sector/block sizes do not affect the hashes.

[5]Where each insertion or deletion is weighted as a difference of one, with changes weighted as three.

Arbitrary bit manipulations may still impact the reliability of rolling hash based techniques, as each individual chunk hash may be manipulated by changing only a single bit in each chunk, with the impact on rolling hashes being dictated by the chunk size and trigger conditions. An alternative strategy, exemplified by the *sdhash* [55] tool, is to reduce the set of features which are used to generate data signatures. In doing so, arbitrary changes to unimportant data, as well as false positives generated from common data blocks, can be better mitigated when correlating similar files. Features are selected by calculating the entropy of data in a 64 byte sliding window. Features with very high entropy (encoding tables) and very low entropy (sparse, repetitive content) are discarded as they do not serve to discriminate between files. The authors derived entropy thresholds empirically to make them realistic. Bloom filters are used to store features, calculating similarity estimates from the overlap of these discriminative features.

Extensive testing [56] of both approaches in tasks involving correlating objects, detecting embedded files, heterogeneous sources and multiple file types, showed that *sdhash* is generally the better performer. *sdhash* is capable of correlating small shared data blocks, while *ssdeep* functions best when large contiguous shared blocks are present in both files. Additionally, the scoring system used by *ssdeep* was shown to generate false positives even at high scores, indicating that it is a poor similarity metric. However, several attacks have been discovered against both approaches [57], with trigger points [58] being exploited in *ssdeep*, and Bloom filter [59, 60] and feature attacks [61] against *sdhash*. These weaknesses would allow many false positives to be generated, circumventing the ability to reliably identify target files automatically.

Both *ssdeep* and *sdhash* are relatively old algorithms, published in 2006 and 2010 respectively. The area of bytewise matching has seen improvements made to both algorithms over time [62], with additional rolling hash based schemes [63, 64], indexing approaches [65, 66], and use of Field Programmable Gate Arrays (FPGAs) to improve performance [67]. However, while there have been other techniques developed for bytewise matching, *ssdeep* and *sdhash* remain popular in the literature, likely due to their support by NIST in the form of official hash sets [68]. Alternative schemes which make use of blocks to build a representation of the file, such as SimHash [69], MinHash [70], mvhash-B [71] and bbhash [72] have limitations, such as high runtime costs, file specific configuration, and a poor sensitivity in duplicate detection [52, 62]. As such, they will not be discussed in depth.

Bytewise approximate matching allows for contraband to be detected with much higher reliability than traditional full file hashing due to bit level manipulation resistance. This robustness is provided at the cost of complexity, however, as the database storage and lookup costs are higher [52]. However this extra expense allows bytewise methods to locate file fragments or media embedded in other files. When detecting similar files, this approach does not work well on compressed files types, such as JPEG, as pixel level manipulations can result in large changes in the binary representation, throwing off byte level approaches [57]. Given that a large use case for known file detection is often based on the analysis of compressed media, this is a significant problem. The next subsection discusses semantic approximate matching, which sacrifices calculation speed for more robust content signatures.

2.3.2 Semantic Approximate Matching

While bytewise approximate matching schemes generate signatures and conduct file comparisons based on the binary representation of a file, semantic approaches make use of knowledge about the structure of a particular file type, generating signatures from the human-facing content. Bytewise methods would be ineffective at identifying a copy of an image which had been saved as a different file type, changing, for example, from a JPEG file to a PNG. Semantic approaches, on the other hand, should be able to identify the transformed image based on its perceptual, semantic, content. While semantic approximate matching can be applied to a variety of file types, the focus of this thesis is on detecting contraband images for forensic purposes and, as such, the semantic approaches discussed will fall under the banner of perceptual image hashing.

Perceptual hashing approaches are inspired by the domain of Content Based Image Retrieval (CBIR) [73], with a wide variety of features which can be used to create a hash which is representative of the visual content. One key aspect of perceptual hashing is the ability to compare signatures for images which are both visually identical and visually similar. This leads to a distinction between *(i)* content preserving manipulations - such as the introduction of noise, compression, scaling, rotation, cropping, and colour, contrast or gamma adjustments, and *(ii)* content changing manipulations - such as moving, removing or adding objects, modifying characteristics structures, textures or colours, and modifying lighting conditions [40]. When used in forensics, this allows for altered contraband images to be detected even when traditional cryptographic hashing and bytewise approximate matching would completely fail.

Hadmi et al. [40] provide an overview of perceptual hashing techniques, using slightly different classifications to the cryptographic; bytewise approximate; and semantic approximate matching classes discussed by Breitinger et al. [52]. The authors identify a typical pipeline for perceptual hashing: *(i)* Transformation - the content is spatially transformed (e.g. colour manipulation, smoothing or rescaling) or undergoes a frequency transformation (e.g. Discrete Cosine Transform (DCT), Discrete Wavelet Transform (DWT)), *(ii)* Feature Extraction - feature vectors are generated from all, or a subset, of features in the spatial or transform domain, *(iii)* Quantization - each component feature is reduced and combined into an intermediate, continuous, perceptual hash vector, which may result in the introduction of fragility to modification, and finally, *(iv)* Compression or Encryption - a fixed length final hash is generated, potentially using traditional cryptographic hashing schemes. The authors note that common distance (dissimilarity) metrics include hamming distance, bit error rate, and the popular similarity metric *Peak of Cross Correlation (PCC)*.

A wide array of features have been used as the basis for generating robust image signatures. Histograms and statistical information about the entire image may be used, providing a high level representation which is insensitive to small, localised, changes in the image. This can take the form of colour histograms [74], texture and edge histograms [75], or frequency domain statistics [76]. Properties of human vision may also be exploited, such as our insensitivity to high frequency changes in an image over a small area, a property which is exploited by JPEG compression [77]. Low frequency properties of an image may be used to derive a perceptual hash [78, 79], which provides robustness to compression artefacts and other content preserving modifications. In a similar vein, coarse image representations may be used, such as low resolution versions of the image, or the average colour value of sub-blocks in the image [80]. Alternatively, invariant relationships in the image may be exploited, such as those found on radial lines projected out from the centre of the image [81]. The latter approach is resistant to rotation and scaling.

In a forensic context, perceptual hashing methods can be used to detect contraband image variants, a prime example being Microsoft's PhotoDNA [82] solution, which has been licensed by a number of service providers for the large scale detection of child abuse media [83]. A further benefit is the ability to compare reduced versions of an image with each other. For example, image thumbnails may be compared to their original image as long as the perceptual hash is appropriately robust to image re-scaling. Additionally, the features chosen by perceptual hashing algorithms provides insight into which elements of an image may be used to generate a reduced representation, particularly when the most popular image compression format, JPEG, makes use of a frequency domain transformation as part of its compression scheme. However,, while identifying images with similar visual content is a non-trivial problem in itself, there are many more requirements of an algorithm before it can be practically employed in a forensic context. The best matching algorithm may be too slow for reasonable use in forensics, or the signatures themselves may contain information which could result in the reconstruction of a contraband image, which would disqualify its use. The next section discusses additional forensics constraints for image signatures in order for them to be feasible forensic solutions.

2.3.3 Forensic Considerations for File Signatures

Digital Forensics investigations deal with large quantities of data, with the computational and memory requirements of a signature generation algorithm being an important aspect of a modern investigation. A given signature generation method may not be fit for forensics purposes, even if it is highly accurate at detecting contraband. Breitinger et al. [52] codify a variety of forensically relevant properties of signature generation tools in order to evaluate them:

Compression: Compression is the ratio between the size of the input file and the representative hash. This is important as lookup databases potentially include millions of contraband signatures, which may make it impossible to store the table in RAM with some schemes, significantly impacting lookup performance.

Runtime Speed: The time it takes to create signatures for individual input files. Large feature extraction and signature creation times may render a technique impractical.

Lookup/Comparison Speed: The speed at which individual items are checked against the known contraband database. A typical investigation may involve a large number of comparisons, with both runtime and lookup rates dictating the scalability of a method.

Sensitivity and Robustness: It is important that an approach be robust against common modifications, otherwise it does not provide many benefits over traditional cryptographic hashes. However, the algorithm must also be sensitive enough to detect partial matches in cases where previously unencountered images are composites or modified versions of a known illegal image.

Contraband databases for child exploitation contain millions of items [45, 47], making memory efficiency, and therefore compression, important. Bloom filters have previously been discussed [45, 53] as a mechanism for reducing the RAM footprint, but it should still be noted that if a solution is perhaps intended to run on a suspect device for the purposes of triage, low system specifications may become a design requirement. Processing speed is also critical at this scale. When traditional cryptographic hashes are stored in a hash table they have the benefit of $O(1)$, constant time, lookups or $O(\log n)$ when stored in binary trees [52], though lookup performance can deteriorate for very large databases [84]. Benchmarking carried out by Breitinger et al. [52] showed that bytewise approximate techniques are slower than traditional cryptographic hashing for both extracting and looking up signatures, while still being considerably faster than perceptual hashing methods. The worst case complexity for approximate matching approaches is quadratic ($O(n^2)$), as it involves each file signature being compared in a pairwise fashion with all signatures in the database. Fortunately, there are mitigation strategies in the literature for perceptual hashing techniques [85], which would enable such approaches to be feasible for real time operations.

The temporal deficiencies of perceptual hashing are traded for increased domain specific robustness to modified images, but they are unable to be used for other forensically relevant correlations, such as detecting embedded or partial files. The results of their evaluation left Breitinger et al. [52] to propose a model for deploying the above methods in a forensic context, in order to best leverage the strengths of each approach. It is suggested that traditional cryptographic hashes be used initially, followed by semantic hashing if no matches are found. The final stage is to perform file carving and bytewise matching to detect fragments and embedded objects.

The goal of this thesis is to reduce the time taken to detect contraband on a device, therefore the above lookup, processing, and memory requirements are critical. This means that, where possible, traditional cryptographic hashes should be used on data subsets, except in cases where robust data signatures are a key aspect of the approach. The next section discusses how reduced file representations have been used in prior work.

2.4 Utilising Reduced Image Representations

It is not always necessary to use all of the available data to achieve a particular task, indeed using reduced representations of a file may achieve the same end result as processing an entire file while being more efficient. A reduced representation may involve simply creating a subset of the data, as with some triage approaches in Section 2.2, or by reducing the data via transformation, as with perceptual hashing techniques in Section 2.3.2. Another approach may be to use file metadata in lieu of file content to make decisions, operating on a higher layer of abstraction. This section discusses various ways in which reduced file representations are used as a proxy for an entire file to achieve a particular task, with literature being sampled from a variety of computing domains.

2.4.1 Reduced File Representations in Digital Forensics

Detecting the presence of contraband on a device is not the only challenge forensic practitioners face. As the purpose of digital forensics is to facilitate legal proceedings, the source of the image, or the integrity/modification status of the image, is often in need of scientific verification. This can be achieved using image and signal processing techniques to detect imperfections introduced in the manufacturing process of the physical device which captures the image [86], however, such approaches can be very computationally expensive, and may still be defeated by advanced forgeries [87].

A body of work has developed which makes use of JPEG header information, as opposed to pixel data, to determine the source device or software used to generate the image, greatly reducing the processing complexity of the task. Quantization tables, the primary source of the lossy aspect of JPEG compression, may be used as a feature for this purpose. While the JPEG standard [77] provides default base quantization tables, which can be scaled to different 'quality factors', digital cameras, and software such as Adobe Photoshop, often make use of their own custom tables. These custom tables allow for the optimisation of image quality based on the characteristics of the camera or software processing method. As a result, they encode information about the source of the image, which can be used to identify the origin [25, 88–90]. The results of this work suggest that while quantization tables are not unique enough to identify a single source, they can serve as a crude mechanism for detecting images which have been processed by software or to reduce the list of possible sources. In some cases quantization tables may be optimised based on the visual properties of the image [89] (dubbed adaptive quantization tables), theoretically providing more discriminating information about the pixel data in that particular JPEG. However, to be useful such adaptive tables would have to be computed by the image creation software and embedded in the image prior to examination.

Quantization tables can be combined with other header features to produce more robust signatures. Gloe [27] provides an analysis of the fusion of quantization tables and ordered data structures, such as file markers, thumbnails and Exchangeable Image File Format (EXIF) metadata, for forgery detection. The primary observation is that the ordering and formatting of elements in the JPEG header can be used as a feature, with different software tools for editing images, or modifying image metadata, behaving differently at this low level. Using this approach, it is possible to identify the software which has been used to modify an image, the circumvention of which requires a high level of programming expertise.

Kee et al. [26] make use of additional features to detect manipulated camera images. Features are extracted from the data contained within quantization tables, EXIF metadata, image dimensions and thumbnails. The additional data produces signatures which were empirically shown to be effective at identifying image provenance for 1.3 million Flickr images, with 62% of signatures allowing the identification of a single camera, 80% to three or four cameras, and 99% to a unique manufacturer, respectively. Several features, such as Huffman tables and EXIF metadata were stored in compressed representations, using only the number of Huffman codes of each length and counts of EXIF fields. EXIF metadata was shown to be the most discriminative feature for the dataset used.

JPEG headers are a rich source of information, but only constitute a small part of the file data, allowing for a very small portion of the file to be used to reduce expensive computation. The above work suggests that small slices of data can be forensically useful at the file level. Indeed, this can be seen in the block based hashing schemes discussed earlier in Section 2.3. This presents an opportunity to exploit such file features to reduce the amount of data which is required to carry out basic forensics analyses, which could provide a means to mitigate a source of forensic bottlenecks.

The next section briefly discusses how metadata based approaches, similar to those discussed above, may be used in the context of email spam detection. The literature in this area may suggest further ways in which reduced file representations may be exploited in a wider forensics context.

2.4.2 Reduced File Representations in Spam Detection

Advances in the field of spam classification led to a change in spammer behaviour around 2005, when a significant shift began from text based spam to image based spam, rendering text based Bayesian classifiers ineffective [91]. Additionally, spammers programmatically generated noisy and randomised images, defeating cryptographic hash based detection [91]. New approaches were needed to identify spam images, with some approaches using Optical Character Recognition (OCR) to extract text to be used in Bayesian classifiers, or by basing features on colour and texture statistics [92]. However, spam detection needs to be scalable to work with high volume email servers, and pixel based techniques are computationally expensive, necessitating the adoption of more efficient approaches.

To enable fast classification, Krasser et al. [93] make use of a subset of image metadata, deriving features from the image dimensions, aspect ratio, file type, file size, image area and compression factor. These features are used to train both Support Vector Machine (SVM) [94] and decision tree classifiers, intended to be deployed as an initial filtering stage in the detection process. Dredze et al. [95] make use of similar features but include more metadata features, such as bit depth and EXIF data, as well as more expensive edge and colour information when training classifiers. However, EXIF metadata features were found to be the most useful, particularly when optimising for speed with dynamic feature selection. Uemura and Tabata [96] design a two stage process, incorporating similar metadata features into a traditional Bayesian filter model, with text analysis as the first stage, and image metadata the second. Liu et al. [97] utilise image header information as part of a triple layer system, where the first stage analyses email header data, the second stage image header data, with the third stage performing costly pixel domain analyses.

An important feature to note is that while metadata based approaches are fast, they may lack some accuracy, resulting in the development of a tiered approach, similar to the hashing process model developed by Breitinger et al. [52]. As such, fast metadata based approaches have the potential to filter out a large portion of data cheaply, such that more robust but expensive methods can be used on what is left. The next section discusses the use of reduced representations in the context of Content Based Image Retrieval.

2.4.3 Reduced File Representations in Content Based Image Retrieval

The header features used in forensics and spam detection contexts are somewhat removed from the pixel content of an image and rely heavily on image metadata. However, it is possible for an image to undergo significant modification while altering little of this metadata. In the field of Content Based Image Retrieval, where image matches are determined based on the content of the media, some approaches directly exploit the fact that modern media files are stored in a compressed format, deriving features from the compression domain and avoiding costly pixel data processing [21]. Such approaches may be applied to both images and video files [98], and are capable of producing average performance on par with pixel domain techniques, while taking less than 15% of the time to extract the necessary features [21].

Most published work in this area focuses on using features of the JPEG file format, which relies heavily on the Discrete Cosine Transform (DCT) for its compression, with quantization tables being applied to the resulting DCT matrices to effect lossy compression. Using these coefficients it is possible to generate histograms of the pixels encoded in the DCT matrix [99–102], which typically correspond to an 8×8 pixel block. These may also utilise texture features encoded in the DCT [99, 100, 102, 103], and are capable of outperforming histogram based pixel domain techniques in retrieval tasks, while reducing processing complexity by an order of magnitude [99, 101]. As global image features, which are derived from the image as a whole, lose fine grained detail and are insensitive to localised changes, the image may be divided into several sub-images to enhance CBIR, which can then be used to generate feature strings, based on properties such as the relationships between the average colour of each sub-image [80, 104]. A different approach, espoused by Schaefer and Edmundson [105], treats compressed JPEG data as a stream of DCT coefficients, which is approximately how the data appears on disk. Features are then derived from the differences between DCT encoded pixel blocks without having to reassemble the DCT matrix.

Header Based Image Retrieval

The compressed domain techniques above operate using the entire image file, sparing computation by avoiding costly operations, but still requiring that the entire file be read and processed to obtain all DCT coefficients. However, it is possible to obtain reasonable retrieval performance by extracting features from the JPEG header alone, while also further reducing computational load.

While the majority of JPEG compression stems from the quantization applied to the DCT matrix, an additional level of compression is provided by using Huffman encoding and differential encoding on the DCT coefficients when saving them to disk. Both quantization tables and Huffman tables appear in the JPEG header, prior to pixel data, and therefore only require a very small fraction of the file to be read for feature extraction. Adaptive, or optimised, quantization tables have been used to rank images for retrieval [106], inexpensively performing on par with the worst of the compressed domain retrieval methods [107]. Similar to how quantization tables may be optimised based on the content of the image [89, 106], Huffman tables may also be optimised to achieve more effective compression. This optimisation, and effective re-compression of the image data, allows these tables to be used as a proxy for image content when deriving image representations, as it will communicate coarse statistical properties of the image. Huffman based retrieval has been explored, using the ordering of elements [23] in the table, or the lengths of the encoding strings [108, 24] as features, achieving relatively good retrieval performance while offering a 30-fold speed improvement over full file compression domain techniques, and 150-fold over pixel domain techniques [24]. Both quantization and Huffman based methods have been utilised as a pre-filtering method for data reduction, with more expensive retrieval methods operating on the remaining subset [22], which retained retrieval performance while greatly reducing query time.

In a forensic context, the cost of pre-processing images, requiring that the entire file be read, is at odds with the intended goal of reducing analysis times. However, Huffman and quantization tables are a potential source of useful forensic information, either when comparing images, or in identifying their source, at low computational complexities. Indeed, a common motivation for using reduced image representations is that they substantially reduce computational complexity for a variety of tasks. Header based methods also allow for small fractions of files to be read, providing an additional reduction in I/O overhead, which can be potentially be exploited in the forensic detection of contraband.

2.5 Review of Chapter

Digital forensics must keep pace with new advancements in technology, however this has not been a trivial task, with the information age leading to the generation of boundless volumes of data across heterogeneous platforms. The failure to cope with this explosion of data in digital forensics has necessitated the use of digital forensics triage, in an attempt to maximise the use of resources, and reduce investigative backlogs. From the literature in Section 2.2, two general approaches to reducing investigative turnaround were discussed: parallel processing, and data reduction. Parallel processing techniques are still fundamentally limited by the read speed of the storage device, while combining the acquisition and processing stages of an investigation. Data reduction methods, on the other hand, reduce both data acquisition times and computational complexity by working with subsets of the evidence, and therefore appear more promising for improving forensic performance. Data reduction approaches essentially represent a trade-off between accuracy and speed, which may have to be accepted if the forensic volume problem is to be overcome.

As contraband detection is a critical part of public sector investigations, and the focus of this thesis, Section 2.3 discussed the various signature generation techniques which can be leveraged for automatic contraband detection. Semantic hashing approaches were discussed as a means of combating the fundamental weakness of traditional hashing when applied to modified files. However, this incurs a computational cost, and does not allow for the detection of file fragments. Bytewise approximate matching methods are suitable for detecting file fragments and small binary modifications, but would prove little use when images have undergone global changes. Future approaches to signature generation should be robust to trivial file modifications, while minimising the computational complexity required for both signature extraction and signature lookups.

Finally, Section 2.4 discussed the use of reduced image representations, where image metadata, headers, or image compression features may be used for a variety of purposes. Certain reduced representations may efficiently effect the data reduction introduced in Section 2.2, while allowing for robust signatures to be generated, by processing only a subset of a file, or by exploiting existing file structures which provide information pertaining to the image. Ideally, such a signature generation approach should also be resistant to small file modifications, but may chose to forgo this in favour of the performance benefits offered by traditional cryptographic hashing these reduced representations.

One final consideration is the prevalence of networked and cloud storage in recent years, which has been identified as a growing concern in forensics [15, 18] as criminals are increasingly utilising cloud storage to store illegal media [13]. This type of forensics is still in its infancy, and necessitates logical, file level, acquisitions [109], as opposed to traditional low level physical acquisitions. An additional concern is that forensics done over a network may take much longer due to bandwidth and network throughput limitations [110]. This scenario should be kept in mind when developing new forensic tools, as cloud storage forensics would similarly benefit from the previously discussed data reduction approaches.

2.6 Conclusions

The volume problem in digital forensics is far from being solved, with additional work being required to bridge the gap between the needs of practitioners, and the reality of the modern investigation. The work in this thesis makes use of reduced file representations which reduce the amount of data to read from disk. File features are chosen carefully to represent a favourable trade-off in terms of uniqueness, false positive rates, and processing speeds. Additionally, as the approaches in this work are based on logical, file level, acquisitions, they should be applicable in other constrained environments, such as networked and cloud storage, where only file level access may be possible. The next chapter discusses the first of three approaches taken in this work, which focuses on the creation of file type specific sub-file signatures.

Chapter 3

File Type Specific Sub-file Signatures

3.1 Introduction

This chapter investigates creating file type specific signatures for contraband detection, where features of a particular file format are used to discriminate between files of that type. Existing work, which makes use of file type specific features, has typically done so for the purposes of identifying the type of a file [28], or for forensically reconstructing carved files [111]. One work makes use of the structure of MP4 files in order to extract high entropy blocks for detecting known video files being transmitted across a network [112], but does so by processing the entire file at the macroblock level and applying the max-hash algorithm [67]. In contrast, the work in this chapter creates signatures which can identify a particular file while processing as little of the file as possible.

Two approaches will be discussed which make use of JPEG and PNG file headers to create forensically viable file signatures. The techniques are inspired by prior work in spam filtering, compressed domain retrieval and header based forensics, as discussed in Section 2.4. While this header based approach may work for many forensically relevant file types, JPEG and PNG were chosen for a proof of concept as they are currently the most popular lossy and lossless image compression formats, respectively.

Existing data reduction approaches in forensics (Section 2.2) make use of disk level random sampling or subsetting heuristics. In contrast, the work presented in this chapter effects data reduction at the file level, allowing for the deterministic processing of all files. Analysing a portion of every file alleviates concerns that files have been missed, while also reducing the amount of data to read from the disk. File headers constitute a very small part of an image file, and typically contain the information required to compress and render the image. The actual portion of a file which needs to be processed to extract such headers, and therefore generate signatures, is evaluated empirically for each file type (Section 3.4.4 and Section 3.5.4).

3.2 Requirements of Sub-file Signatures

Several requirements and useful properties of effective contraband signatures have been identified from prior work in evaluating forensic signature generation [52]:

1. **Uniqueness / Discriminating Power**: In order for a file signature to be used effectively as a means of detecting contraband, it should aim to be unique. This effectively means that the false positive rate should be as close to zero as possible. However, the use of less discriminating signatures may be mitigated with a hierarchical processing approach, where more accurate techniques are used to evaluate potential hits.

2. **Robustness**: While not a strict requirement, signatures should ideally be tolerant of trivial file modifications, such as flipping an arbitrary bit or introducing content preserving modifications.

3. **False Negatives**: If no modification to the original file has been introduced, the file should be detected with 100% accuracy. This is the minimum bar required to compete with traditional full file hashing.

4. **Generation Speed**: In order for a sub-file signature to be useful, signature generation should be significantly faster than processing the entire file (as with full file hashing). This represents the effective reduction in processing time, and therefore places an upper bound on the effect the approach has on reducing backlogs.

5. **Lookup Speed**: Fast signature extraction is useless if it is not paired with equally fast database lookups. Ideally, this should be constant time (O(1)) or logarithmic time (O(log n)), which is the case for traditional hashing approaches [52], as the slower similarity hashing schemes have been shown not to keep up with disk read speeds [11].

6. **Compression/Signature Length**: Signatures must be as small as possible, and ideally be a fixed length. This potentially allows databases to be stored in RAM to facilitate fast lookup speeds, while allowing many records to be stored before the database gets too big for main memory.

The evaluation of the signature generation techniques described in this thesis takes each of these properties into account, with a particular focus on generation speed, lookup performance, and discriminating power. The baseline that sub-file approaches are compared against is the full file hashing approach, as this is the standard method used when detecting contraband media.

Dataset	**File Type**	**No. Files**	**Mean Size**		**Median Size**	
			KiB	**MiB**	**KiB**	**MiB**
Flickr 1 Million	JPEG	1000000	124 KiB	0.12 MiB	117 KiB	0.11 MiB
Govdocs	JPEG	109233	336 KiB	0.33 MiB	79 KiB	0.08 MiB
Govdocs Optimised	JPEG	109228	326 KiB	0.32 MiB	76 KiB	0.07 MiB
Govdocs PNG	PNG	108885	1426 KiB	1.39 MiB	344 KiB	0.34 MiB
Bing (Collected)	PNG	6469	381 KiB	0.37 MiB	132 KiB	0.13 MiB

Table 3.1 Details of the datasets used in this chapter.

3.3 Description of Datasets

In order to evaluate the proposed techniques, it was necessary to gather large, real world, datasets. This was important as files which are generated from a single source may not be sufficiently representatives of real world files. No actual contraband images were used, however the proposed approaches should generalise to images containing any content.

Details of the datasets used in this chapter are provided in Table 3.1. The Flickr 1 Million [113] and Govdocs [114] datasets are both publicly available, while Govdocs Optimised and Govdocs PNG are modifications of the original. The Govdocs Optimised dataset uses optimised Huffman tables generated using the *jpegtran* [115] tool, while the Govdocs PNG dataset re-encodes the original JPEG images to the PNG format using the python PIL library [116]. Some files failed to process, resulting in slightly smaller modified datasets. The Bing dataset was a supplementary dataset collected by issuing queries to the Bing search API. Details of how the dataset modifications and Bing collection were produced are available in Appendix A.1. No modifications in the binary or pixel domain were made to the images, except those specified above.

Optimising Huffman tables in JPEGs could potentially be done in a number of ways, however the method employed here, using *jpegtran*, employs the standard approach found in the libjpeg library [117], from the creators of the JPEG specification. Therefore there is a reasonable expectation that most optimised Huffman tables will be created using the same approach. Using Python's PIL library to encode JPEGs to PNG means that the transformed dataset will be homogeneous in nature, with each image possessing similar header and encoding properties. While data found on a typical hard drive is unlikely to be homogeneous in this way, this dataset makes the features of each image less distinctive, making signature generation more challenging. If the partial file approach in this chapter performs adequately on such a dataset it can be expected to perform well in real world scenarios, such that this compromise is acceptable.

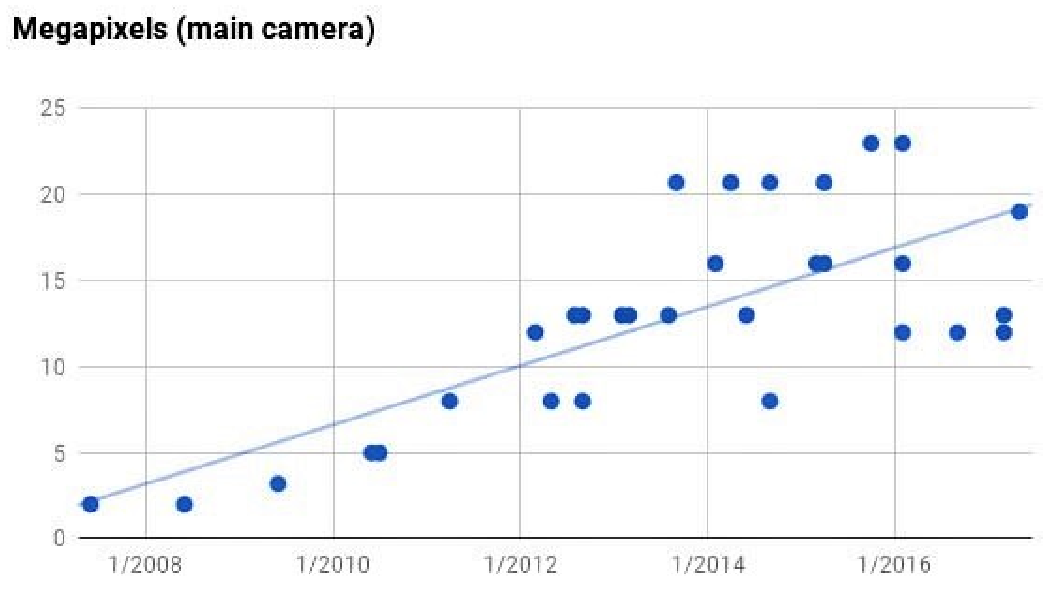


Fig. 3.1 The increase in the resolution of mobile phone cameras over time. Sourced from: https://petapixel.com/2017/06/16/smartphone-cameras-improved-time/

3.3.1 File Size Considerations

It should be noted that the file sizes of the datasets are much lower than may be expected in a real investigation involving contraband. The Flickr 1 Million dataset in particular is very small due to its source, the Flickr website, which is an image hosting platform. As it is in their best interest to reduce file sizes for hosting purposes, these images have been processed to reduce file sizes. The Bing dataset was collected from the general Web, meaning that many images were chosen specifically for Web consumption, and are therefore likely to be on the smaller end of the scale. Finally, Govdocs is a collection created from government websites, and contains a wide range of file sizes, from tens of bytes to tens of mebibytes. The mean file size is skewed upwards due to a small portion of very high resolution images from NASA, however the median file size is very small.

Over the last decade the typical photograph has increased dramatically in size, partially due to the increasing pervasiveness of high speed broadband and WiFi, but also due to advances in camera technology. Figure 3.1 depicts the trend for the resolution of mobile phone cameras to increase over time. A brief analysis of 500 photos taken with the author's Samsung Galaxy S6, which utilises a 16 megapixel camera[6], showed a mean file size of

[6]24bit sRGB JPEGS at 72dpi, with a resolution of 5312×2988

4.5MiB. It is likely that most images produced by cameras, mobile or otherwise, in the last 10 years should have file sizes on the order of mebibytes, rather than kibibytes.[7].

Effectively, the performance benchmarks carried out in this thesis are likely to represent the worst case performance scenario for sub-file hashing when compared to full file hashing. Sub-file approaches only require a tiny fraction of a file to generate a signature, while full file hashing reads every byte of the file. The larger the file, the smaller the proportion of the file which has to be read by the sub-file hashing technique. This means that using smaller than average files results in a less pronounced performance gap between sub-file and full file approaches than may be anticipated in a real investigation. If the relative performance of sub-file signatures is adequate for these small file sizes, their utility can only be expected to improve on a dataset of modern high resolution photographs.

3.4 Sub-file PNG Signatures From Early File Features

This section describes a technique for identifying known images encoded in the Portable Network Graphics (PNG) [118] format based on signatures which are created using only information from approximately 1% of the file. Features are extracted from early portions of the file, focusing on elements which are used to render the image. This approach allows for a highly distinct signature to be created and reduces the quantity of data needing to be processed by 99%. The PNG format is used on over 70% of websites [119], appearing more frequently than the JPEG format, making it a good target for forensics processing efforts. Additionally, as the PNG format is lossless it is a better container choice for animated/artificial images with large homogeneous areas, as JPEG compression introduces obvious artefacts.[8] This work on PNG signatures has been published at the 2017 ADFSL Conference [1].

The remainder of this section will discuss the PNG format, in order to better understand potential sources of signature data in the header, subsequently discussing the features selected for signature generation in this work. The approach is then evaluated against the criteria in Section 3.2, finishing with a brief closing discussion. Supplementary materials, including code snippets, are found in Appendix C.

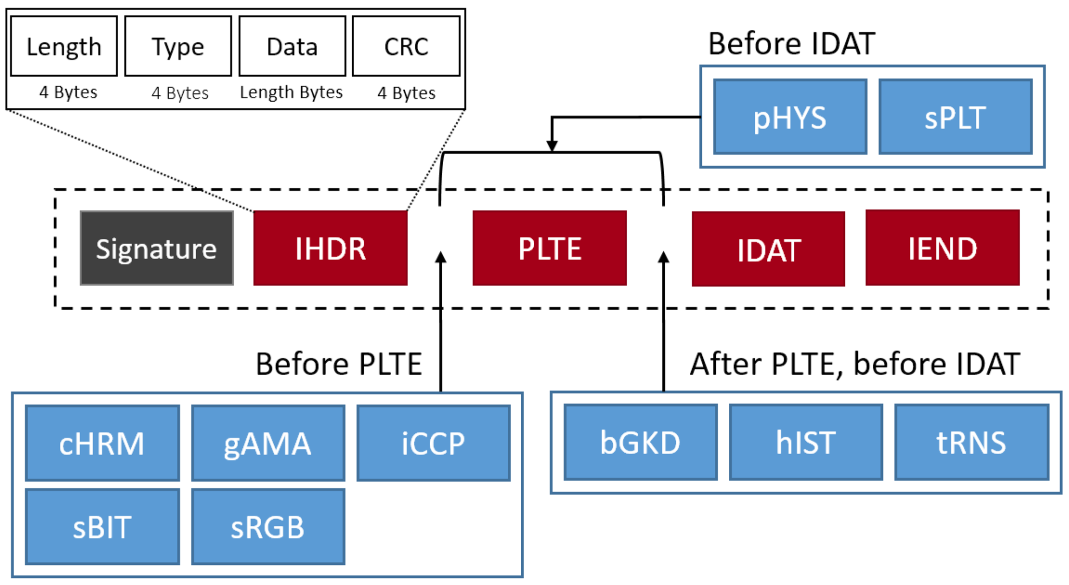


Fig. 3.2 The layout of a PNG file. Critical chunks in red (within dotted lines), ancillary chunks in blue, with textual and timestamp chunks omitted for clarity.

3.4.1 The PNG File Format

The general layout of a PNG file is shown in Figure 3.2. PNG files begin with an 8 byte signature, containing the ASCII bytes for ‘PNG’, as well as various line ending and transmission integrity bytes. All file content thereafter, both metadata and compressed pixel data, is stored in a ‘chunk’ structure.

Chunks are self-contained, storing their own length, data payload, and a checksum to detect corrupted data. Four letter ASCII labels are provided to easily distinguish between chunk types, which are referred to throughout this section.

Labels beginning with upper case characters, depicted in red in Figure 3.2, refer to ‘critical’ chunks, which form the minimal set which an encoder and decoder should support. ‘Ancillary’ chunks, depicted as blue, need not be supported, and may be safely ignored, or omitted, without preventing the image from being rendered. However, such omissions mean that the image may not appear as originally intended, potentially displaying without transparency and backgrounds, or rendering with a skewed colour spectrum.

There are 18 chunk types defined in the international standard [118], with 4 critical chunks and 14 ancillary chunks. The critical chunks are: `IHDR`, containing header metadata, which is required immediately after the signature; `IEND`, required as the final chunk to

[7]Based on estimates using the online tool: https://toolstud.io/photo/megapixel.php The tool underestimates the size of the Galaxy S6 photos, and suggests that 5 megapixels JPEGs should be around 1MiB in size.

[8]This may be particularly relevant in the UK and other countries which also classify artificial child abuse images as illegal, thus making it of interest in such investigations.

complete the file; IDAT, one or more of which store compressed pixel data; and PLTE, which stores the colour palette and is only required for the indexed colour mode.

Ancillary chunks in the specification correspond to colour space information (cHRM, gAMA, iCCP, sBIT, sRGB), pixel dimensions and aspect ratio (pHYs), suggested palette (sPLT), miscellaneous information for the suggested background (bKGD), colour histogram (hIST), transparency information (tRNS), textual information (iTXt, tEXt, zTXt), and timestamp information (tIME). Additionally, custom chunk types can be included by the encoder for use with specific applications, as with Adobe Photoshop's proprietary chunks. With the noted exception of the beginning and end chunks, there are only loose constraints placed on the order or location of chunks in the file. Most chunks are required to be present before the first IDAT chunk (see Figure 3.2), with the exception of textual information, which may be present at any point in the file prior to IEND.

PNG supports five colour modes, which include greyscale and truecolour, both with additional modes which include an alpha channel, and the indexed colour mode. Colour modes are set in the IHDR, with optional reference colour points (cHRM) and profiles (ICCP) in other chunks. Transparency is supported either by means of the alpha channels or by preselected colours which indicate transparency.

Compression of pixel data is achieved by per scan line prediction and the DEFLATE compression standard, which is stored in the zlib format [120]. This data is contained in one or more IDAT chunks in the PNG.

3.4.2 Extracting PNG Signatures

Determining which features were to be used for signature generation involved a manual survey of the chunks in the PNG specification, but also required some empirical work. Several features are not used very often in practice, and it was therefore necessary to determine which header chunks appear consistently in real world use.

One difficulty in carrying out this work was the lack of available large scale PNG datasets, with most image retrieval and forensics datasets focusing heavily on images of the JPEG format. To address this, a collection of 6469 PNG images (2.36 GiB) was collected from the Bing search engine, as described in Appendix A.1. This provided a set of real world PNGs from a wide variety of websites and software sources. However, the Bing dataset was relatively small, and a thorough analysis necessitated a much larger corpus. The Govdocs PNG dataset was created to allow analysis at scale, converting the original JPEG images to the PNG format using the Python Pillow 3.1.1 [116] library. Three images which failed to convert, and a single corrupt PNG, were discarded. The PNG dataset (148 GiB) is larger than its original JPEG counterpart (35 GiB), with file sizes ranging from 170 bytes to 38.2MiB. Diversity in the dataset makes it easier to generate highly discriminative signatures, as different software tools and varied encoding parameters produce different low level features. As the PNG dataset is produced by the same encoder, its properties are homogeneous, meaning that it is effectively a worst case scenario for analysing the discriminating power of the sub-file signature.The selected features, and their associated chunks, are discussed in detail below.

Header Features

The initial intention was to leverage specific features with potentially high discriminating power, such as colour histograms (`hIST`), colour palettes (`PLTE`) and low resolution image scans from interlaced PNGs (early `IDAT`). These features, when present, essentially provide built-in coarse image representations, much like those used by CBIR methods (Section 2.3.2). However, none of these features were used frequently in the heterogeneous Bing dataset (7.2% paletted images and 2.9% interlaced, with no instances of the histogram chunk), such that a more general approach was taken.

The PyPNG [121] python module was used to extract features from PNG chunks prior to the first `IDAT` chunk. This was achieved by using the *Preamble* method of the PNG *Reader* class, which processes image metadata from a common subset of PNG chunk types. The chunk types considered, and their associated features, are provided in Table 3.2. When discussing header features in this section, it is in reference to these chunks. Derivative features, such as the number of planes or the alpha flag, are omitted, as their information is contained in the colour type. Code snippets are provided in Appendix C.1.

Ideally, features should only be extracted from the subset of chunks which contribute to the rendering of the image, ancillary or otherwise. Such features are safe from arbitrary tampering without affecting the way the image displays. Notable omissions for the header features include those containing textual information, timestamps, ICCP colour profiles, standard RGB colour space flags, and proprietary Adobe Photoshop chunks, none of which are obtained using the above python method.

As the `IHDR` chunk is required to be in every image, it was evaluated in isolation from other metadata chunks, serving as a minimal header baseline. To facilitate inter-image comparisons, all features were concatenated into a single string, which allows for simple string equality to determine whether two images have the same feature vector. As the PyPNG function used simply returns a dictionary of objects, the order of items is not preserved as it appears in the file.

Chunk Order

Gloe [27] noted that image encoders for the JPEG format are not necessarily consistent with the order in which they include metadata structures, such that the order of elements

Feature	**Chunk**	**Description**
File Signature	N/A	‘Magic number’ from the file.
Height	IHDR	Image height in pixels.
Width	IHDR	Image width in pixels.
Bit Depth	IHDR	No. of bits per sample or palette index.
Colour Type	IHDR	Colour mode flag (See Section 3.4.1).
Filter Method	IHDR	Filter method byte, standard only uses 0 for adaptive.
Interlacing	IHDR	Image interlacing byte, 0 if no interlacing, 1 for Adam7 interlacing.
Compression Method	IHDR	Compression method byte, PNG standard only uses 0 for Deflate.
Palette	PLTE	1–256 RGB palette values if present, otherwise `None`.
Background Colour	bkGD	Default background colour, otherwise `None`.
Transparency	tRNS	Simple transparency alpha value, single colour value, or `None`.
Gamma	gAMA	Four byte floating gamma value.
Unit is Metre	pHYs	1 byte flag indicating if the unit of measurement is metre.
X-Axis Pixels per Unit	pHYs	X-axis aspect ratio or metric size information.
Y-Axis Pixels per Unit	pHYs	Y-axis aspect ratio or metric size information.
Significant bits	sBIT	Stored the original number of significant bits for lossless recovery.

Table 3.2 PNG Header features and their associated PNG chunks.

can be used as a discriminating feature. As noted previously with regards to chunk placement constraints (see Section 3.2), this is also true in the PNG specification [118]. The `IHDR` and `IEND` chunks must be first and last respectively, however, other chunks have loose, or no, ordering constraints. For example, the `gAMA` and `sBIT` chunks must appear before both the `PLTE` and `IDAT` chunks but could appear in either order.

To determine the potential effectiveness of chunk orders as a feature, all chunk types were aggregated into an ordered list to two depths in the file: ***i)*** All chunks in the file, and ***ii)*** Chunks up to the first `IDAT` chunk. By varying how far into the file is processed, it is possible to determine how much of the file must be analysed for this feature to be useful, if at all. All chunk types, including proprietary and textual metadata chunks, were included, with subsets not being considered due to the initial experimental performance.

Again, to facilitate simple equivalence comparison, and integration with other features, ordered lists are transformed into strings. In this case the order of chunks is preserved as part of the feature vector, such that any change of the chunk order in the file is captured by this feature.

Small Block Data Hashes

PNG header chunks are immediately followed by an `IDAT` chunk, which contains compressed image data. This makes it probable that the first data block of the file contains all of the information necessary to extract the signature, except in cases where there are many metadata blocks in the file header. A small portion of scan data is included as a feature by calculating `SHA256` hashes of the first n bytes of the first `IDAT` chunk, for varying sizes of n. This effectively provides a hash of the first few scan lines of the image.

Additionally, traditional cryptographic hashes were calculated from partial files to act as a baseline for comparison and to determine the relative merits of analysing the file structure. In this case, n bytes from the beginning of the file are used to generate hashes using `SHA256`.

When the value of n is less than, or equal to, the length of the hash digest it would produce (32 bytes), the raw data is included in the signature instead of the hash digest, saving a small amount of needless processing.

IDAT Length

The size of each `IDAT` chunk normally corresponds to the size of the memory buffer used by the encoder [118], though they need not be of a constant length. As the beginning of each chunk stores its own length only the first few bytes are required to obtain the size of a chunk. This means that the length of the first `IDAT` chunk, immediately following the header metadata, may be used as a proxy for the likely buffer size of the encoder, and as a discriminating feature.

The length of the first `IDAT` chunk was obtained by inspecting the *atchunk* property of the PyPNG*Reader* class immediately after calling the *Preamble* method. This returns a tuple, the first of which is the length of the chunk.

Combining Features

As all features above can be easily represented as a string, simply concatenating individual features together, in an ordered fashion, allows for a single signature which can be compared quickly. This also allows the signatures for known files to be looked up in the same way that traditional cryptographic file hashes are utilised, with O(1) complexity from a hash table.

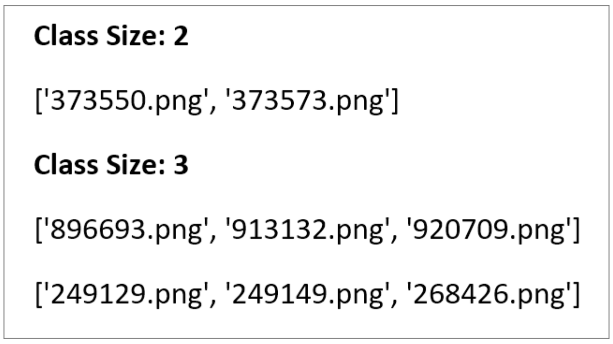
Equivalence Classes

Equivalence classes are generated by recording a list of all images which have identical feature strings. The number of files in a given list corresponds to the size of the equivalence class. This provides a granular measure of how unique a given signature is and allows for the identification of groups which possess the same signature. A class size of 1 indicates that the signature is unique for all files, a class size of 2 contains 2 files which possess the same signature, and so on. An example of hypothetical equivalence class groups is provided in Figure 3.3.

If 95% of the items in a dataset were grouped into equivalence classes of size 1, this would mean that 95% of file signatures are unique. Class sizes larger than 1 can be seen as a form of clustering, where equivalent items are grouped together. Large class sizes, e.g. 100, indicate that there are many images which produce the same signatures, likely due to some shared source (such as camera or software) or shared property among all images in that class (e.g. all using the same encoding tables and metadata structures). When evaluating a signature most of the dataset should be in a class size of 1 for high discrimination.

Offset Acquisition

In order for the above information to be derived the required data has to be read from the storage media. As such, the closer to the beginning of the file the above features are located, the smaller the proportion of the file which has to be analysed. The `IDAT` is the anchor point for all of the above features, as all header features must appear before it, and scan data appears immediately after it. As such, the byte offset of the first `IDAT` chunk is



Class Size: 2

['373550.png', '373573.png']

Class Size: 3

['896693.png', '913132.png', '920709.png']

['249129.png', '249149.png', '268426.png']

Fig. 3.3 Example equivalence classes. One class of size 2, and two classes of size 3. No file may appear in more than one equivalence class.

used to evaluate how much of the PNG file needs to be processed for these signatures to be generated.

3.4.3 Evaluation of Discriminating Power

An evaluation of the discriminating power of each feature is provided in isolation, before exploring the potential of combining features. The best case scenario for each feature is that it generates a unique signature for each file, i.e., an equivalence class of 1 for all images. Timed benchmarks, comparing the Sub-file approach with full file hashing, is provided in Section 3.4.4.

Chunk Orders

There is a wide range of possible chunk types, with the potential for individual applications to include proprietary chunks. The order in which these chunks appear may be used as a feature to discriminate between images. Table 3.3 shows the results of creating equivalence classes of all chunk types for the entire file.

Even utilising all chunks in the file, this feature performs poorly across both datasets, with only 17.9% unique signatures for Bing images, and 0.1% for Govdocs PNG images. Chunk ordering in the Govdocs PNG dataset barely has any discriminating power due to the homogeneity of how the files are constructed. In some cases JPEG metadata, such as ICCP profiles were retained during PNG conversion, but such incidences do not provide much information in the absence of additional ancillary chunks. The Bing dataset has more variety due to additional metadata blocks and encoding differences, though there are still only a small number of critical chunks included in the specification, limiting the usefulness of this feature.

The performance of chunk orders falls further if only chunks up to the first `IDAT` are considered, with the number of unique signatures dropping to 2.4% on the Bing dataset. These results suggest that the majority of the discriminating information is provided by the number of `IDAT` chunks included in the file, as well as any metadata chunks which are interleaved with scan data, or found after it.

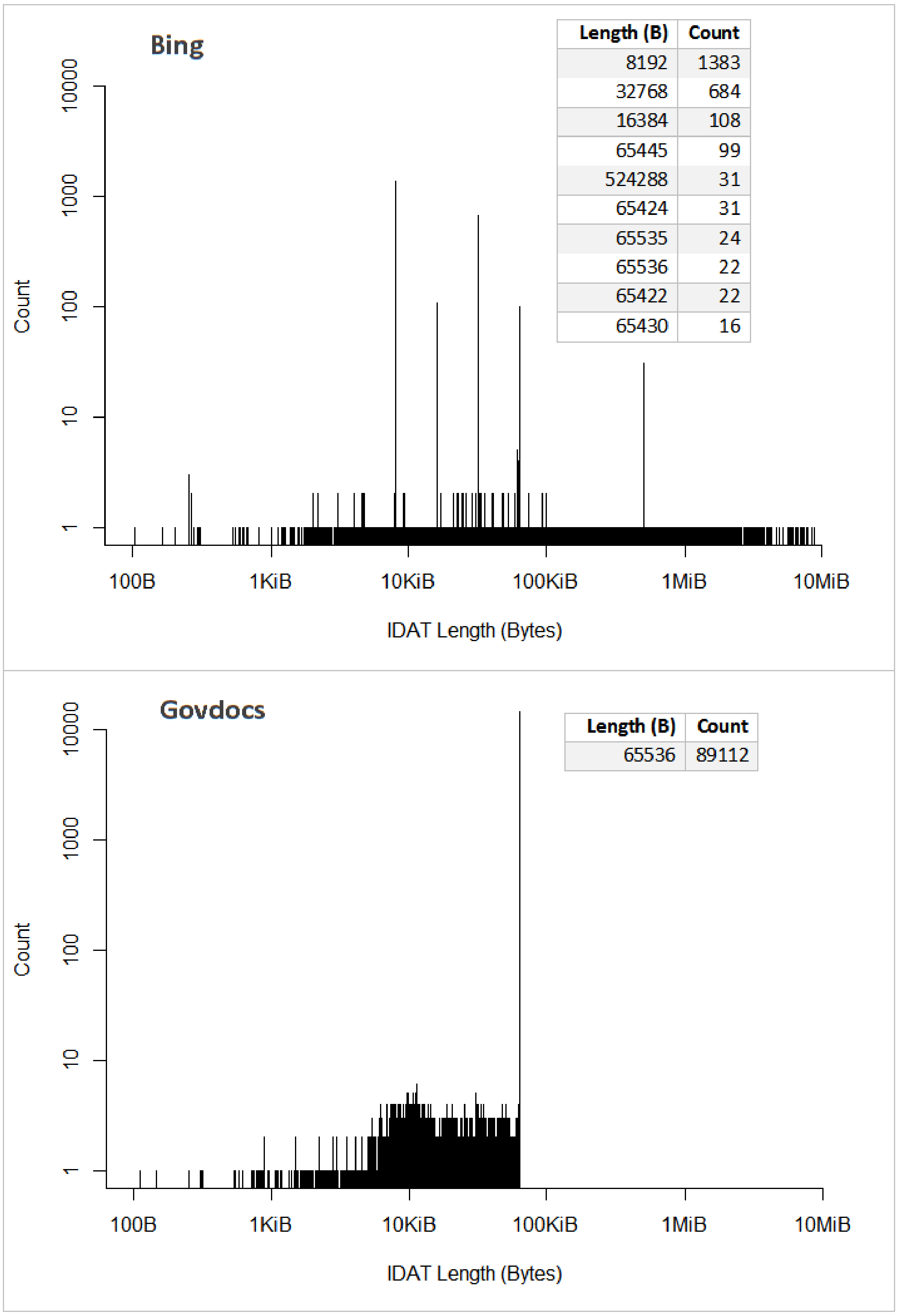
	Equivalence Class Size No. Images (% of Images)				
	1 (Unique)	2-5	6-10	11-100	>100
Bing					
All Chunks Order	1159 (17.9%)	1017 (15.7%)	329 (5.1%)	959 (14.8%)	3007 (46.5%)
IDAT Length	3907 (60.4%)	130 (2.0%)	14 (0.2%)	245 (3.8%)	2175 (33.6%)
Govdocs PNG					
All Chunks Order	121 (0.1%)	364 (0.3%)	402 (0.4%)	4341 (4.0%)	103657 (95.2%)
IDAT Length	13501 (12.4%)	6266 (5.8%)	6 (<0.1%)	0	89118 (81.9%)

Table 3.3 Equivalence classes for chunk type orders and IDAT lengths for Govdocs PNG and Bing.

IDAT Length

The length of the first `IDAT` chunk can be used to gain information about the number of potential data chunks in the PNG, as well as information about the encoder, without needing to analyse the entire file. The performance of this feature varies wildly between datasets (Table 3.3), with 60.4% unique signatures for the Bing dataset, but only 12.4% for the Govdocs PNG dataset. As the Govdocs PNG dataset was produced using a single encoder, the behaviour of this feature on this dataset essentially reflects the behaviour of the Pillow library. In this case, the encoder limits the length of `IDAT` chunks to 64KiB, with larger files being divided into multiple chunks of this size or less. Unique signatures are produced by files which are smaller than this upper limit, with the distribution depicted on the bottom of Figure 3.4.

In the Bing corpus, `IDAT` lengths range from 104 bytes to 8.77MiB, the distribution of which is provided in the top half of Figure 3.4. These results indicate that many encoders opt not to limit chunk sizes, which is possible as the maximum chunk size in the specification is $2^{31} - 1$ bytes (2GiB) [118]. Large spikes in the Bing distribution are likely caused by the use of the same encoding library, or those using similar, content independent buffer presets.



Length (B)	Count
8192	1383
32768	684
16384	108
65445	99
524288	31
65424	31
65535	24
65536	22
65422	22
65430	16

Length (B)	Count
65536	89112

Fig. 3.4 The distribution of IDAT lengths for both datasets, plotted on logarithmic axes.

	Equivalence Class Size No. Images (% of Images)					
Bing	1 (Unique)	2	3	4	5	>5
IHDR Only	4638 (71.7%)	482 (7.4%)	153 (2.4%)	116 (1.8%)	50 (0.8%)	1032 (15.9%)
Header Features	5106 (78.9%)	444 (6.9%)	135 (2.1%)	92 (1.4%)	65 (1.0%)	629 (9.7%)
Govdocs PNG						
IHDR Only	27059 (24.9%)	7626 (7.0%)	4143 (3.8%)	2868 (2.6%)	2230 (2.0%)	64959 (59.7%)
Header Features	27059 (24.9%)	7626 (7.0%)	4143 (3.8%)	2868 (2.6%)	2230 (2.0%)	64959 (59.7%)

Table 3.4 Equivalence classes for the IHDR and complete PyPNG preamble features.

The `IDAT` chunk length is not a suitable standalone feature, as it is sensitive to the encoder used. However, it performs well enough to be used in combination with other features. Additionally, this feature could be useful in future work as part of a process to identify particular encoders or applications which created a given PNG, similar to prior work on JPEGs [25, 88–90].

Header Feature Distinctness

While many of the features described in Table 3.2 have a relatively narrow range of values, with several being composed of a single byte, their combination provides potential to discriminate between images.

The equivalence classes generated by using features extracted by the PyPNG preamble method are given in Table 3.4. Using all features in Table 3.2, 78.9% of the Bing dataset possessed a unique signature. Less than 10% of images were in a class of five or greater, with the largest class consisting of 76 images, meaning that there are relatively few large clusters of equivalent signatures. Using the same encoder in the transformed Govdocs dataset, header features are still unique 25% of the time, which is perhaps higher than expected. Performance does not suffer much on the Bing dataset when only the `IHDR` chunk, which must be present in every PNG, is considered, with no measurable difference being made on the Govdocs PNG dataset.

While these features are not enough to uniquely identify images in the dataset, they are reliably found immediately after the PNG file signature and contribute a good deal of discriminating information. As such, they prove good candidates for combination with additional features. Additionally, it should be noted that not all ancillary chunks which are provided in the specification are processed by PyPNG, and, as such, performance may

IDAT Block Size (B)	**Equivalence Class Size** No. Images (% of Images)					
	1 (Unique)	2	3	4	5	>5
Govdocs PNG						
8	72981 (73.4%)	13932 (14.0%)	5895 (5.9%)	3304 (3.3%)	2305 (2.3%)	10468 (9.6%)
16	107257 (98.6%)	1152 (1.0%)	252 (0.2%)	84 (<0.1%)	45 (<0.1%)	95 (<0.1%)
32	108670 (99.8%)	86 (<0.1%)	42 (<0.1%)	20 (<0.1%)	5 (<0.1%)	62 (<0.1%)
64	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)
128	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)
256	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)
512	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)
1024	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)
2048	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)
4096	108671 (99.8%)	86 (<0.1%)	42 (<0.1%)	24 (<0.1%)	0	62 (<0.1%)

Table 3.5 Equivalence classes for varying block sizes of `IDAT` chunk data SHA256 hashes for the Govdocs PNG dataset.

be improved slightly by including these additional chunk types, at the cost of fragility to arbitrary chunk data.

Small Block IDAT Hashing

Small portions of the scan data may be used to discriminate between images. Equivalence classes for blocks of varying sizes for the Govdocs PNG dataset are provided in Table 3.5, where the block size is the number of bytes, n, which are hashed from the beginning of the first `IDAT` chunk. In some cases, the block size is larger than the size of the content of the first `IDAT` chunk, in which case the actual length of the chunk is used for the hash. However, this did not occur frequently, with only 200 instances when n = 4KiB. Both datasets performed similarly, as such, Bing data is omitted here.

In this experiment, small values of n were sufficient to produce highly distinct signatures. As few as eight bytes of data produce a unique `SHA256` hash digest for 73.4% of images in the dataset, with no substantial benefit beyond 32 bytes, where 99.8% of signatures were unique.

An examination of equivalence classes for the larger block sizes shows that duplicate hashes are caused by images which have large contiguous arrays of bytes with the value zero. As PNG is a lossless format, it can represent large areas of contiguous, identical, values without artefacts, both for solid colour and transparent backgrounds. In these cases, when the same background value is used, matching hashes will be generated as the value of n may be too small to reach the differentiating data in the foreground. That is, while

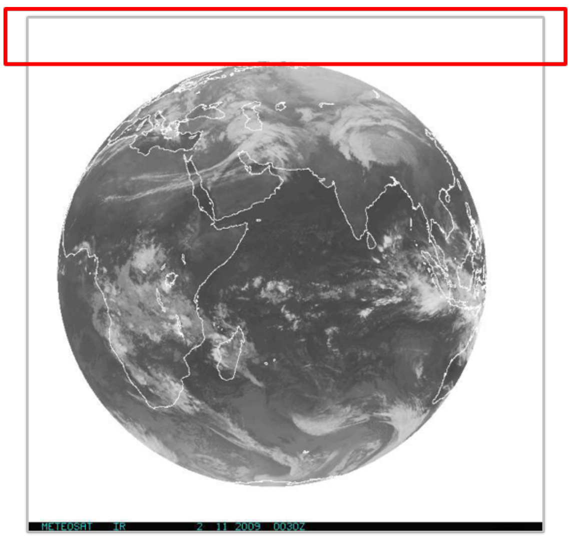


Fig. 3.5 Example PNG likely to generate false positives with small IDAT hashes. This is due to homogeneous areas at the top of the image, highlighted by the red box.

this technique is generally effective, it may produce false positives with logo style images. An example image which may generate false positives using this approach is provided in Figure 3.5.

Small Block File Hashing

Rather than hashing the whole file, or data contained solely in the `IDAT` chunks, the first *n* bytes of the file were hashed to serve as a baseline. Table 3.6 shows the results of these small block hashes. Again, performance was consistent across datasets.

Block hashes prove to be very distinct, though this strategy was not as successful as `IDAT` hashing for very small block sizes. Hashing the first 8 and 16 bytes in the file created the same signatures for all images in the dataset. When the block size reaches 32 bytes, file hashes perform identically to features extracted from the `IHDR` chunk (Table 3.4), as they likely contain the same content.

Hashing the first 2048 bytes of the files produced duplicate hashes more often than 16 bytes of `IDAT` data. A full disk sector (4KiB) still performs worse than 32 bytes of the first `IDAT` on Govdocs PNG but produces unique values for the Bing corpus. The relative performance of `IDAT` and File hashes is depicted graphically in Figure 3.6.

Multiple Feature Aggregation

With the exception of 4KiB file hashing on the Bing dataset, no single feature tested distinguished all distinct images. To further improve file discrimination, multiple feature

Start of File Block Size (B)	**Equivalence Class Size** No. Images (% of Images)					
	1 (Unique)	2	3	4	5	>5
Govdocs PNG						
8	0	0	0	0	0	108885 (100%)
16	0	0	0	0	0	108885 (100%)
32	27059 (24.9%)	7626 (7.0%)	4143 (3.8%)	2868 (2.6%)	2230 (2.0%)	64959 (59.7%)
64	104814 (96.3%)	898 (0.8%)	408 (0.4%)	356 (0.3%)	215 (0.2%)	2194 (2.0%)
128	105034 (96.5%)	846 (0.8%)	381 (0.3%)	320 (0.3%)	175 (0.2%)	2129 (2.0%)
256	105034 (96.5%)	846 (0.8%)	381 (0.3%)	320 (0.3%)	175 (0.2%)	2129 (2.0%)
512	105447 (96.8%)	778 (0.7%)	345 (0.3%)	276 (0.3%)	160 (0.1%)	1879 (1.7%)
1024	105474 (96.9%)	762 (0.7%)	339 (0.3%)	276 (0.3%)	155 (0.1%)	1879 (1.7%)
2048	105474 (96.9%)	762 (0.7%)	339 (0.3%)	276 (0.3%)	155 (0.1%)	1879 (1.7%)
4096	108670 (99.8%)	84 (<0.1%)	42 (<0.1%)	20 (<0.1%)	0	69 (<0.1%)

Table 3.6 Equivalence classes for SHA256 hashes of the first *n* bytes of the file, ignoring file structure. Govdocs PNG dataset.

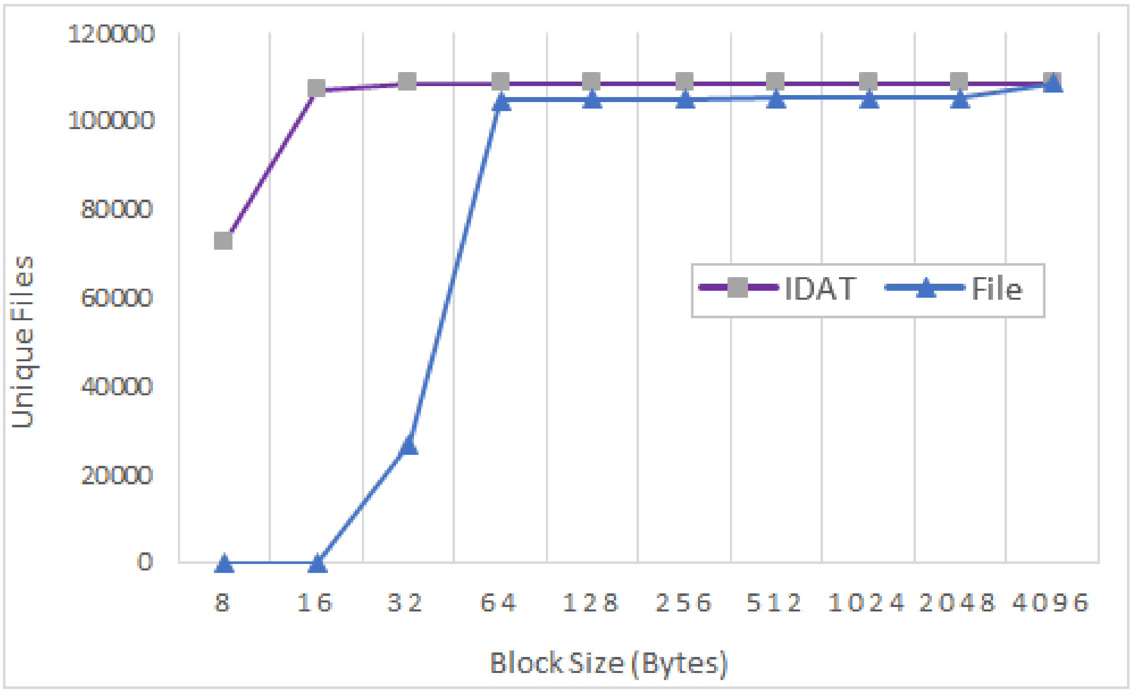


Fig. 3.6 The number of unique IDAT and File hashes for each block size for the Govdocs PNG dataset.

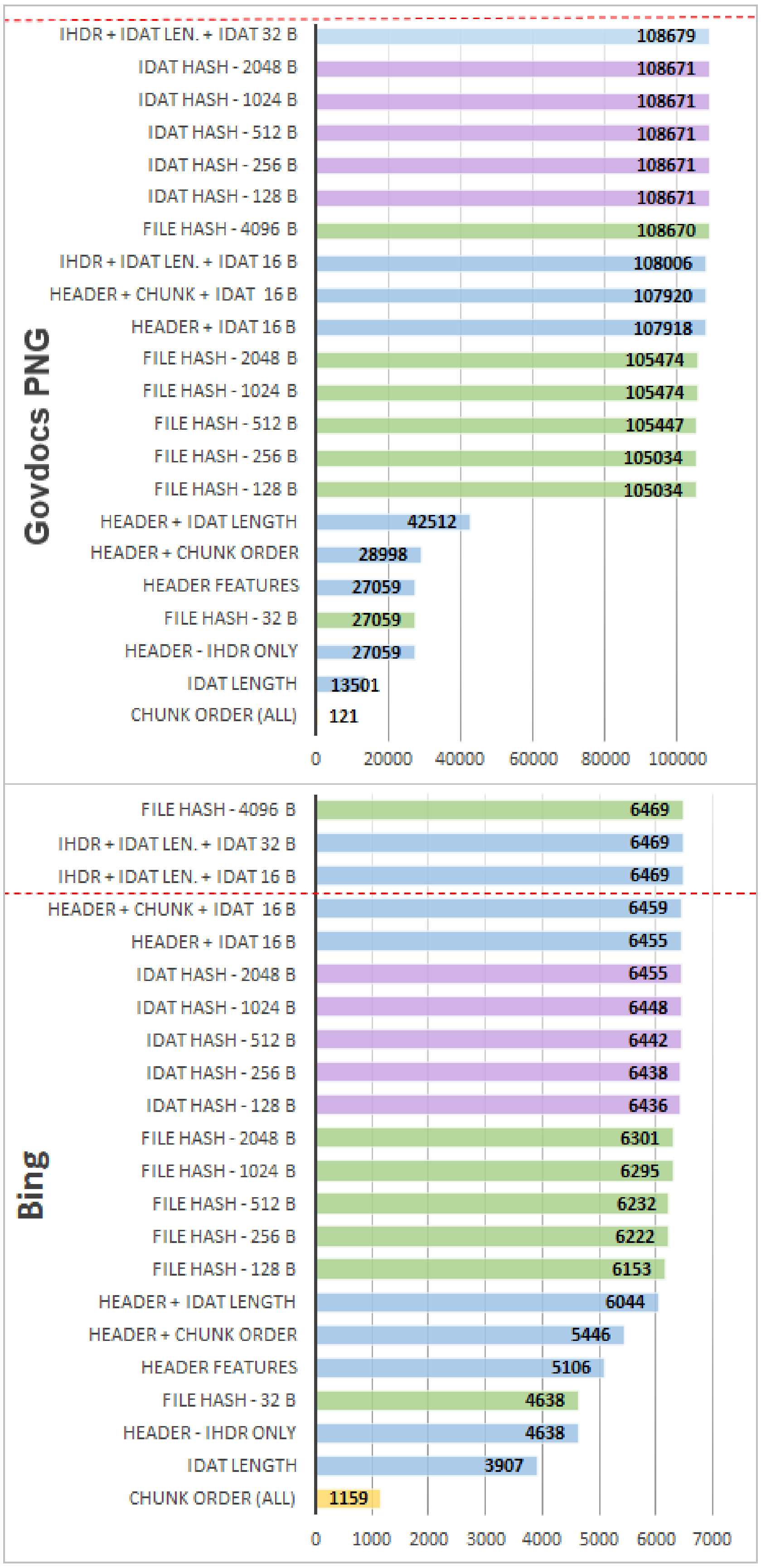


Fig. 3.7 The number of unique signatures for each feature and dataset. The dotted line indicates where all unique images have unique signatures.

strings were concatenated together to form a single signature. A graphical overview of the relative effectiveness of single and combined features for both datasets is provided in Figure 3.7.

As noted above, chunk type ordering was not a strong feature by itself, and provided minimal utility when combined with additional features. However, chunk orders can be acquired at a very low expense, while extracting other, more discriminative, features.

The best performance is achieved when combining small `IDAT` data block hashing with `IHDR` features and the length of the first `IDAT` chunk, with only a small number of bytes required from the scan data. This combination was able to distinguish between images in the dataset just as well as 4KiB file hashes, while providing resistance to arbitrary modifications of non-critical metadata, such as textual information. It also has the benefit of only processing data which is required of every PNG file, which is essential for rendering the image. False positives produced using this combined approach are typically caused by homogeneous areas of pixel data at the top of the image, as previously discussed for Figure 3.5, while others were caused by nearly identical images which were not cryptographic hash matches.

The signatures produced are small, between 35 and 42 bytes using 16B of `IDAT` data, and between 51 and 58 bytes using 32B of `IDAT` or the `SHA256` digest. This signature size achieves a good level of compression, with the length being smaller than larger cryptographic hash digests, such as SHA512 (64 bytes). As the PyPNG signatures are variable length, it may be worthwhile hashing the signature strings with an algorithm such as SHA256 to produce smaller, fixed length, digests.

3.4.4 Lookup Performance Evaluation

Algorithm 1: PyPNG Signature

```
Input: PNG File
Output: PNG Signature
signature = String(readIHDR());
while chunk.type != IDAT do
    chunk.type = nextChunk();
end
signature += String(chunk.length);
signature += String(chunk.data[0:32]);
return signature
```

Based on the findings in this work, it is possible to conclude that highly distinct PNG signatures may be derived from the mandatory `IHDR` chunk when combined with information found at the start of the first `IDAT` chunk. This contains encoding information about the

	IDAT Offset (Bytes)								
Percentile	10%	25%	50%	75%	90%	95%	97.5%	99%	99.999%
Bing	37	74	113	956	2753	2774	8826	33,812	2,587,401
Govdocs PNG	37	37	37	37	37	2658	2658	2658	30181

Table 3.7 Percentile offsets (in bytes) for the beginning of the first IDAT chunk in both the Govdocs PNG and Bing datasets.

image, as well as dimensions, and a proxy for the memory buffer size, with a tiny portion of scan data to rule out very similarly encoded images with the same dimensions. Using this method, a tiny fraction of the file may be used to create a signature for a PNG file, while also ignoring easily stripped, non-essential, metadata modifications. Pseudocode for signature creation using these features is provided in Algorithm 1.

What follows in this section is an analysis of the potential performance improvements of this approach over full file hashing, beginning with quantifying how much of a typical PNG file needs to be acquired to create these signatures.

Typical IDAT Offsets

The location of the first `IDAT` chunk in a PNG is the point at which the header information stops, and the pixel data scans begin. It therefore serves as a good indicator of how much of the file needs to be read with the proposed Sub-file approach. Its location varies with the quantity of metadata included in the file. If there are few chunks included between the `IHDR` and first `IDAT` block, it can appear very early in the file, with the smallest offset in both datasets being 37 bytes. When many chunks are included, pixel data can start deep in the file, with 2.5MiB of data being present before the `IDAT` in the worst case in the Bing corpus and 103KiB in Govdocs PNG. However, as noted, the transformed Govdocs corpus does not contain much ancillary metadata as it is a transformation from JPEG, with the vast majority of `IDAT` blocks starting at 37 bytes. Table 3.7 shows the IDAT byte offsets at various percentiles for both datasets.

The distribution of `IDAT` offsets in the Bing corpus is very long tailed, with a median of just 113 bytes, and mean of 4158 bytes. As the mean file size is 381KiB, the features discussed can be acquired by only reading **approximately 1%** of the file for images in the Bing dataset, with far less being required for Govdocs PNG (0.01%). 96.9% of Bing images had an `IDAT` marker appear within the first 4096 bytes (99.9% for Govdocs PNG), with 60% of files (92% for Govdocs PNG) requiring a mere 160 bytes to reach the `IDAT`. As modern storage media make use of 4KiB disk blocks, this means that most PNG signatures were acquired by reading a single disk block. This is a very small overall proportion of the relatively small files in these datasets, such that early file features represent a substantial reduction in disk overhead, while also producing a highly discriminative signature.

While this is a good level of discrimination, even on homogeneous datasets (99.8% unique signatures), it is not accurate enough to avoid false positives at the million image scale.[9] However, the utility of this approach lies in being able to filter out large portions of data, with potential hits being verified with more accurate, and costly, methods, such as traditional cryptographic hashing or similarity hashing. In doing so, the amount of data to be processed can be reduced by two orders of magnitude, from 100GiB to 1GiB, or less. Assuming this technique is substantially faster, it can still be used in a tiered processing system for data reduction. The next section discusses timed benchmarks carried out to evaluate speed improvements.

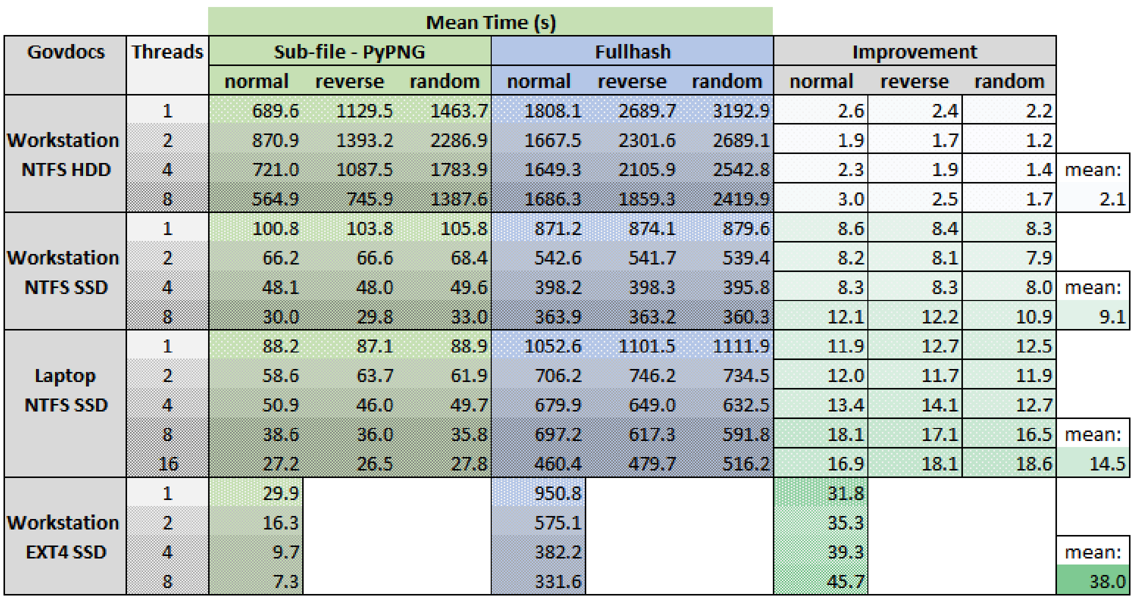
Timed Benchmarks

To quantify the potential speed improvements, several timed benchmarks were performed using the larger Govdocs PNG dataset, directly comparing the PNG sub-file approach in Algorithm 1 with the traditional full file hashing approach. The SHA256 algorithm was used for all cryptographic hashing, as this is the smallest hash from the popular cryptographic hashing families which has not been compromised.[10] However, as cryptographic hashes are inexpensive to calculate compared to fetches from storage media, this added overhead should not have any bearing on the results. As signatures in both the sub-file and full file hashing schemes are strings, they can both be stored in hash tables for O(1) lookup. The benchmarks, therefore, primarily focus on the I/O bottleneck of the device. All code is written in Python 2.7, with code snippets available in Appendix C.1.

Two computers were used to evaluate performance: *i)* a workstation - i5-4690k, 16GiB DDR3 RAM, Western Digital Red 4TB HDD, Crucial MX300 525GB SSD, and *ii)* laptop - i7-5500U, 8GiB DDR3 RAM, Samsung 840 EVO 500GB SSD (OS). Storage benchmarks are provided in Appendix B.1. Both machines were tested with different thread counts and file read orders to assess the impact on storage throughput and relative performance. Three file orderings were used: *i)* normal, provided by the Python `os.listdir` command, which gives the ordering used by the file system, *ii)* the reverse ordering of the normal ordering, and *iii)* random ordering, created by generating random sorts of the normal list. Files were copied in the same order to all drives with no fragmentation. Benchmarks were carried out on Windows 10 64bit, with memory caches being cleared between runs using the EmptyStandbyList tool [122]. Benchmarks were run automatically in sequence using a script, as in Appendix C.1, with cache clearing meaning that sequential runs should not

[9]Which is the scale of frequently used contraband datasets.

[10]Both MD5 and SHA1, while commonly used in forensics, have been shown to be weak against attack.



		Mean Time (s)									
Govdocs	Threads	Sub-file - PyPNG			Fullhash			Improvement			
		normal	reverse	random	normal	reverse	random	normal	reverse	random	
Workstation NTFS HDD	1	689.6	1129.5	1463.7	1808.1	2689.7	3192.9	2.6	2.4	2.2	
	2	870.9	1393.2	2286.9	1667.5	2301.6	2689.1	1.9	1.7	1.2	
	4	721.0	1087.5	1783.9	1649.3	2105.9	2542.8	2.3	1.9	1.4	mean:
	8	564.9	745.9	1387.6	1686.3	1859.3	2419.9	3.0	2.5	1.7	2.1
Workstation NTFS SSD	1	100.8	103.8	105.8	871.2	874.1	879.6	8.6	8.4	8.3	
	2	66.2	66.6	68.4	542.6	541.7	539.4	8.2	8.1	7.9	
	4	48.1	48.0	49.6	398.2	398.3	395.8	8.3	8.3	8.0	mean:
	8	30.0	29.8	33.0	363.9	363.2	360.3	12.1	12.2	10.9	9.1
Laptop NTFS SSD	1	88.2	87.1	88.9	1052.6	1101.5	1111.9	11.9	12.7	12.5	
	2	58.6	63.7	61.9	706.2	746.2	734.5	12.0	11.7	11.9	
	4	50.9	46.0	49.7	679.9	649.0	632.5	13.4	14.1	12.7	
	8	38.6	36.0	35.8	697.2	617.3	591.8	18.1	17.1	16.5	mean:
	16	27.2	26.5	27.8	460.4	479.7	516.2	16.9	18.1	18.6	14.5
Workstation EXT4 SSD	1	29.9			950.8			31.8			
	2	16.3			575.1			35.3			
	4	9.7			382.2			39.3			mean:
	8	7.3			331.6			45.7			38.0

Fig. 3.8 Summary of benchmark results for the Sub-file PyPNG approach and Full File Hash (Fullhash) for different configurations. Improvement factors on the right indicate how many times faster the Sub-file approach is. Only normal ordering was used for EXT4 runs as ordering did not make a difference.

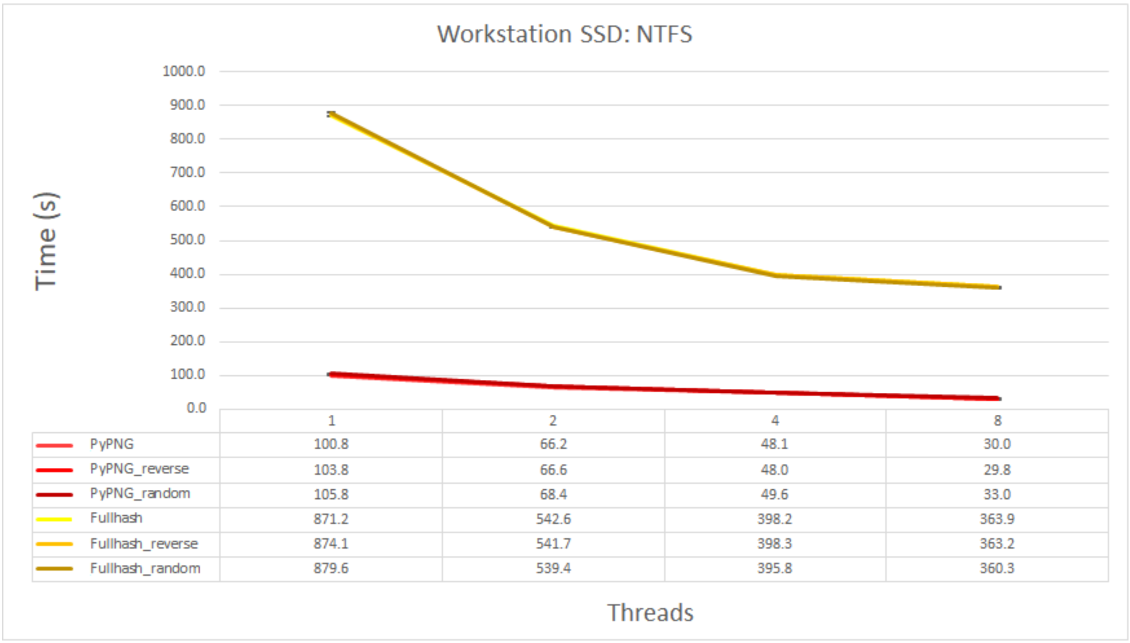
have an impact on each other. Experiments were repeated three times on the desktop and 10 times on the laptop, which is more susceptible to background processing interference.[11] Reported times represent the mean total time to extract the signature and store it in a python dictionary, with the times to enumerate files with `os.listdir` not included. A summary of the benchmark results is provided in Figure 3.8, with a breakdown of more detailed results thereafter.

SSD Performance

On solid state media, there is a clear performance advantage for the sub-file PyPNG approach, graphed for the workstation in Figure 3.9 and the laptop in Figure 3.10. Both approaches benefit from increasing the thread count from 1 to 8, with a smooth downwards curve in total time taken as threads are added on both devices. However, the file ordering made little to no difference, which is to be expected given the nature of SSDs, which have no mechanical seek times or spinning platters.

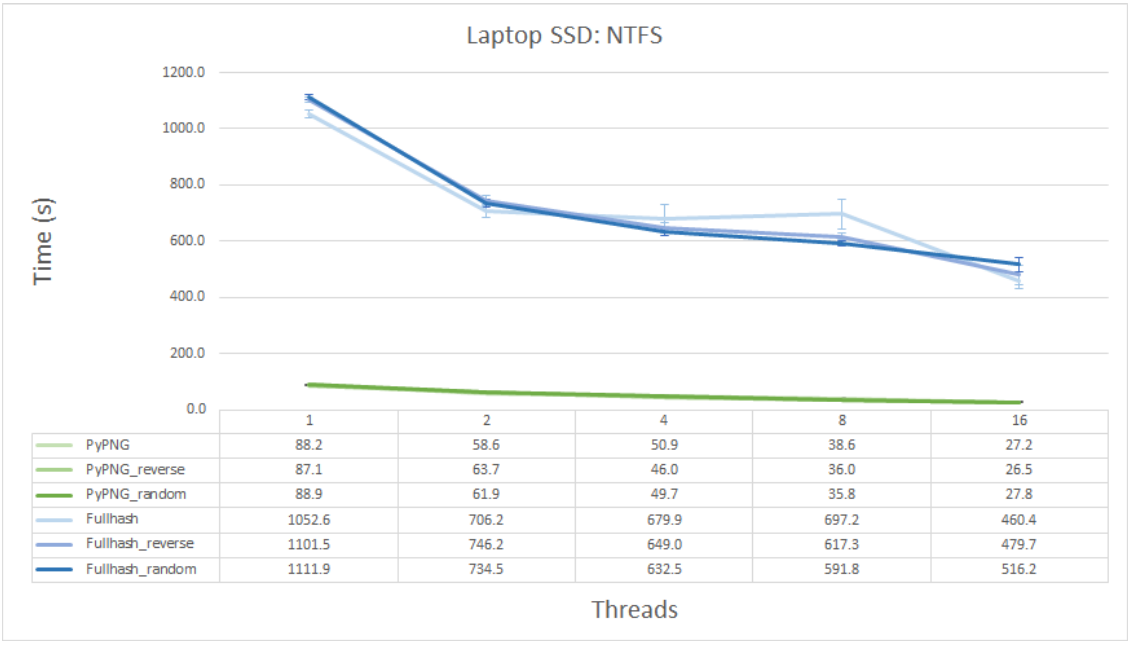
At 8 threads, the sub-file approach proved to be 12$\times$ faster on the workstation, and 18$\times$ faster on the laptop, as summarised in Figure 3.8. The large gulf between approaches is not caused by CPU bottlenecks, as full file hashing only reached 40% CPU utilisation on

[11]This is due to the smaller number of CPU cores on the laptop, as well as the fact that the drive is running the operating system.



	1	2	4	8
PyPNG	100.8	66.2	48.1	30.0
PyPNG_reverse	103.8	66.6	48.0	29.8
PyPNG_random	105.8	68.4	49.6	33.0
Fullhash	871.2	542.6	398.2	363.9
Fullhash_reverse	874.1	541.7	398.3	363.2
Fullhash_random	879.6	539.4	395.8	360.3

Fig. 3.9 A comparison of the relative performance of the Sub-file and Full File Hashing approaches on the workstation's NTFS formatted SSD. Mean time over three runs. Error bars are provided, but are very small and not clearly visible.

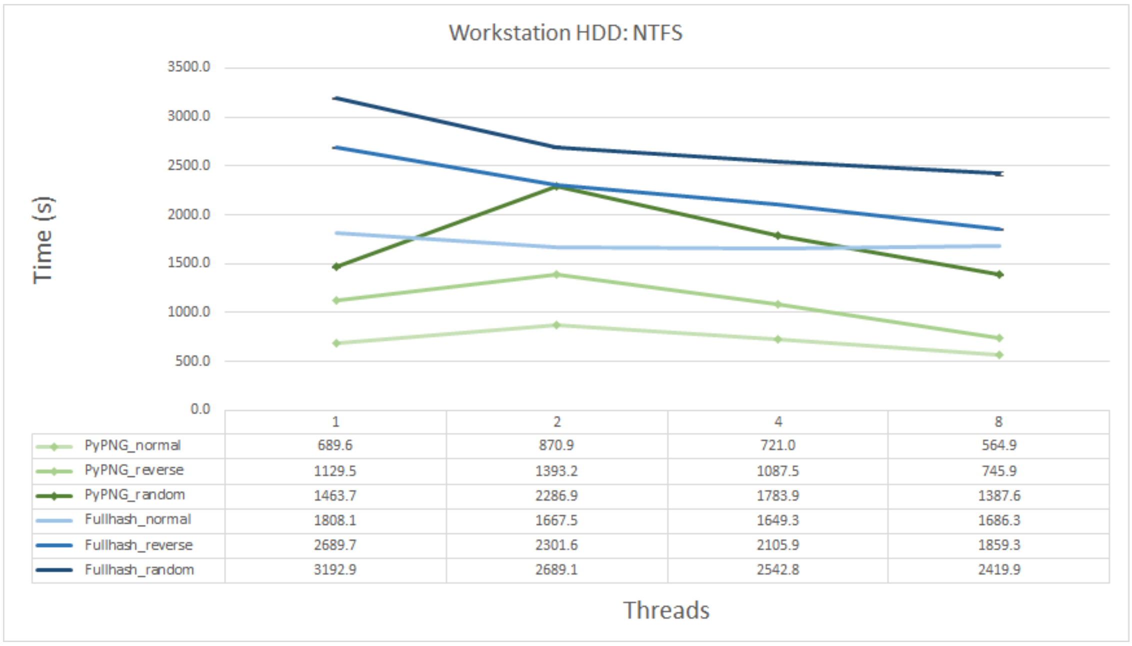


	1	2	4	8	16
PyPNG	88.2	58.6	50.9	38.6	27.2
PyPNG_reverse	87.1	63.7	46.0	36.0	26.5
PyPNG_random	88.9	61.9	49.7	35.8	27.8
Fullhash	1052.6	706.2	679.9	697.2	460.4
Fullhash_reverse	1101.5	746.2	649.0	617.3	479.7
Fullhash_random	1111.9	734.5	632.5	591.8	516.2

Fig. 3.10 A comparison of the relative performance of the Sub-file and Full File Hashing approaches on the laptop's NTFS formatted SSD. Mean time over ten runs. Errors bars provided, but are only substantial for the normal Fullhash runs.

the workstation, and 70% on the laptop. Rather, the effective data reduction of reading 1% of the file results in substantial benefits, despite the relatively low small block performance of both SSDs when compared to their sequential throughput (Appendix B). The differences between the relative performance on the workstation and the laptop are likely due to the slightly better random 4KiB read performance at low queue depths on the laptop SSD.

HDD Performance



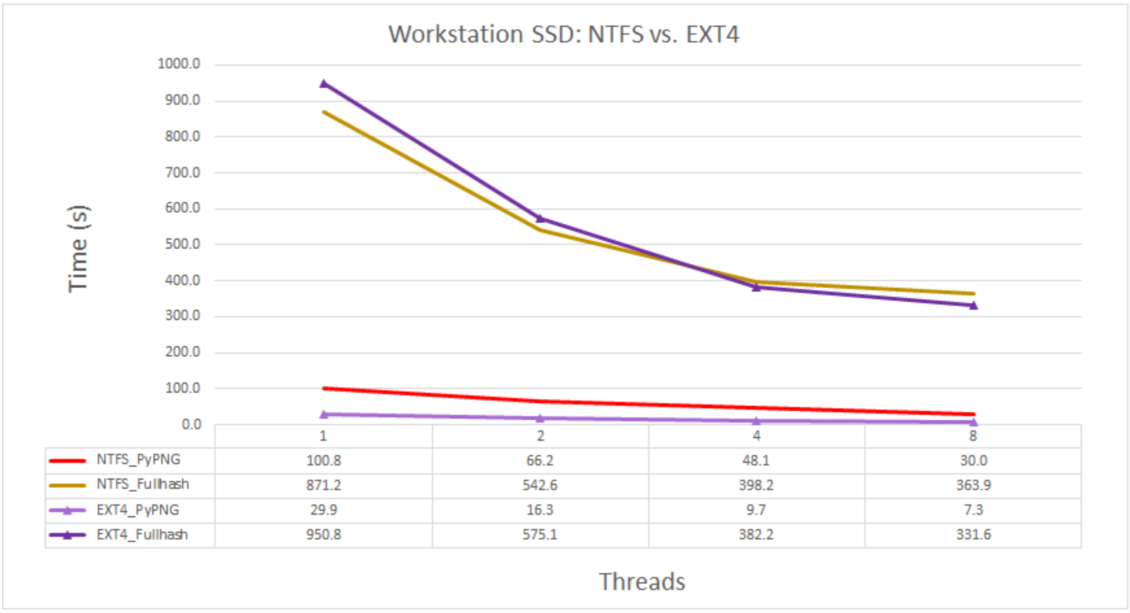
	1	2	4	8
PyPNG_normal	689.6	870.9	721.0	564.9
PyPNG_reverse	1129.5	1393.2	1087.5	745.9
PyPNG_random	1463.7	2286.9	1783.9	1387.6
Fullhash_normal	1808.1	1667.5	1649.3	1686.3
Fullhash_reverse	2689.7	2301.6	2105.9	1859.3
Fullhash_random	3192.9	2689.1	2542.8	2419.9

Fig. 3.11 A comparison of the relative performance of the Sub-file and Full File Hashing approaches on the workstation's NTFS formatted hard disk drive. Error bars are provided, but are very small and not clearly visible

Performance on the workstation's hard drive, depicted in Figure 3.11, indicates that the behavioural characteristics of the underlying storage media can have a large impact. On the hard disk, random ordering was substantially slower than the normal read order, with reverse times falling in the middle. As the hard disk has to physically spin to each data region, introducing mechanical delays, this is to be expected. However, this also has the effect of decreasing the relative benefit of the sub-file approach, which holds a reduction of $3\times$ for the normal ordering at 8 threads, but only $1.7\times$ for the random ordering. The same mechanical latencies also effectively increase the total time it takes to fetch a small data blocks, with very low random 4KiB read performance relative to the sequential throughput of the hard disk (Appendix B). For this reason, while full file hashing at 8 threads was $4.6\times$ slower on the HDD relative to the SSD, the sub-file approach suffers at $19\times$ penalty.

Despite this drastic cut in performance, the sub-file approach is still appreciably faster on this dataset.

File System Overheads



	1	2	4	8
NTFS_PyPNG	100.8	66.2	48.1	30.0
NTFS_Fullhash	871.2	542.6	398.2	363.9
EXT4_PyPNG	29.9	16.3	9.7	7.3
EXT4_Fullhash	950.8	575.1	382.2	331.6

Fig. 3.12 A comparison of the relative performance of the Sub-file and Full File Hashing approaches across the NTFS and EXT4 file systems on the workstation's SSD. Error bars are provided, but are very small and not clearly visible

An additional experiment carried out using the workstation's SSD with Ubuntu 15.04 shows that the file system also plays a large part in overall performance. The same experimental setup was used as with Windows, with the exception of how the cache was cleared[12], while making use of bash scripts instead of shell scripts to automate the benchmarks. One limitation of this approach is that the Linux `ntfs-3g` driver is used for the NTFS volume, which introduced small overheads for NTFS, while EXT4 ran via the native driver. The impact of this, and other overheads, is discussed in depth in Section 3.6.3.

Figure 3.12 depicts the relative performance of the same SSD formatted with both NTFS and EXT4. At 8 threads EXT4 performs better for both approaches, netting a 10% performance increase for full file hashing, with a much larger increase of 410% for the sub-file approach. This indicates that there are potentially substantial file system overheads when looking up files indexed by the file system, which in this case is simulated by the `ntfs-3g` driver. By reducing these overheads, the sub-file approach receives a

[12]Cached were cleared on Ubuntu using the command:
`sudo sh -c ‘echo 3 > /proc/sys/vm/drop_caches’`

dramatic reduction in overall file access times, and therefore a substantial performance improvement, resulting in a massive 45× performance increase over full file hashing on EXT4 at 8 threads. This allowed over **100,000 files to be processed in less than 8 seconds**, when compared to the the 332 seconds for full file hashing. A brief benchmark comparing the sub-file approach to reading a single 4KiB block for each file was also carried out on EXT4. This benchmark showed that the sub-file signature performance is typically 80%–90% of the speed of reading the first 4KiB of each file, though this decreases slightly at 16+ threads.

3.4.5 Tiered Filtering Benchmarks

As this sub-file approach is not 100% accurate, it would be prudent to confirm positive hits in some manner, as to avoid false positives during an investigation. One way of doing this is to verify a positive hit using full file hashing, as depicted in Figure 3.13. This confirmation approach is similar to the hierarchical models used in spam filtering [97], where more expensive processes are carried out after cheap initial passes.

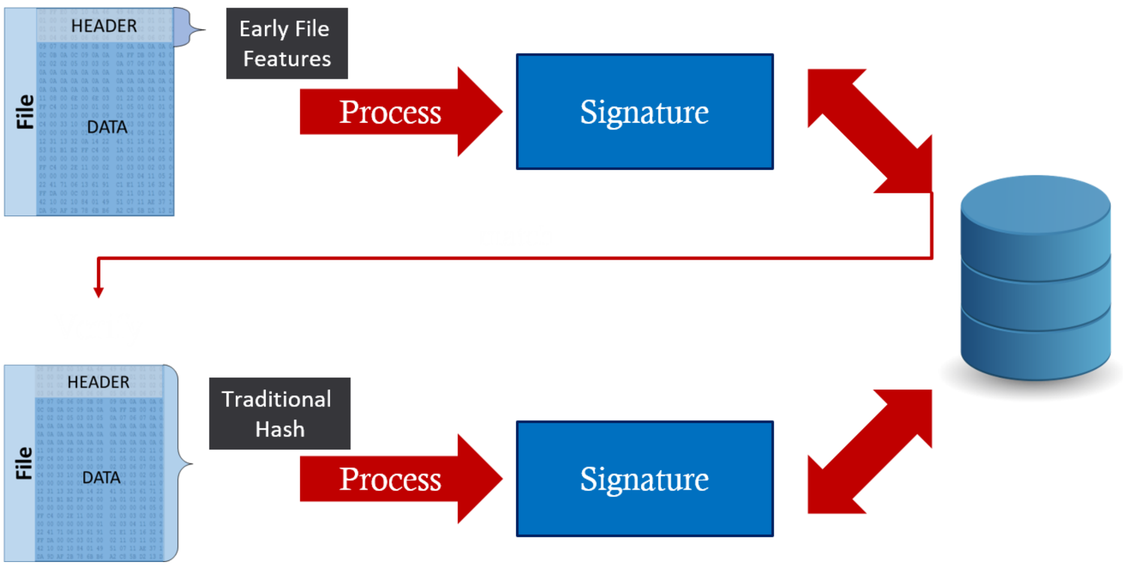


Fig. 3.13 PNG confirmation hashing. Verify positive hits using the more expensive full file hashing approach.

Table 3.8 shows the impact of full file confirmation hashing on the sub-file PNG approach for various hit rates. 1% hit rate means that 1% of files in the corpus generated a positive hit. The higher the hit rate, the more files must be fully hashed, meaning that the ratio of contraband plays a part in how performant this approach is. A hit rate of 1% does not make much of an impact on processing speeds, with the sub-file approach retaining over a 10× performance increase over hashing all files on SSDs, and 3× on the HDD. At

Device	**Threads**	**Sub-file Time (s)**	**Hit Rate (% Contraband)**			**Fullhash Time (s)**
			1%	10%	33%	
Workstation NTFS HDD	8	564.9	581.7 (2.9×)	733.5 (2.3×)	1126.4 (1.5×)	1686.3
Workstation NTFS SSD	8	30.0	33.7 (10.8×)	66.4 (5.5×)	151.2 (2.4×)	363.9
Laptop NFS SSD	8	38.6	43.2 (10.7×)	84.7 (5.4×)	191.9 (2.4×)	460.4
Workstation EXT4 SSD	8	7.3	10.6 (31.4×)	40.4 (8.2×)	117.7 (2.8×)	331.6

Table 3.8 Fullhash confirmation impact on Sub-file PNG. 1%, 10% and 33% of corpus assumed to generate a positive hit, respectively. Calculated by adding the appropriate percentage of Fullhash time to the Sub-file approach, and therefore ignores the benefits of caching. Relative improvement factors in brackets.

a hit rate of 33%, which is likely extreme, the sub-file approach is still between 50% and 180% faster than hashing all files.

One important thing to note is that this confirmation can be done in idle time after the main processing has finished. This would allow for a trade-off of accuracy and initial results, allowing this approach to retain very high effective triage speeds.

3.4.6 Outcome

The contributions of this section are three-fold. The first is a breakdown of the features of the PNG format which appear early in the file, and an identification of those features which can be used in signature generation. The second is an evaluation of the discriminative power of PNG header and small data block features, with a view to creating signatures for unique objects. Finally, this research demonstrates the potential for highly accurate, and efficient, file signature creation which may be used as part of a pre-filtering scheme to reduce investigation processing times. It has also been shown that various storage media and file system technologies impact the relative performance of this sub-file approach.

By processing the header of a PNG, it is possible to generate signatures which are 99.8% unique, even on homogeneous datasets. As only 1%, or less, of the file is needed, this effects a data reduction of 100-fold, at the cost of some inaccuracy. Images encoded using the same encoder, and those using solid backgrounds are the most likely to generate false positives, however these can be verified afterwards using full file hashing, while still retaining most of the performance benefit.

The best case performance increase was 3× on an NTFS HDD, 18× on an NTFS SSD, and 45× on an EXT4 SSD, with an associated reduction in CPU load (see Appendix C.2.1). These results suggest future storage technologies could be similarly exploited, with the potential for very fast sub-file detection using NVMe/M.2. devices which have drastically improved random 4KiB read performance compared to SATA based devices. Higher small block throughput would allow this method to approach its theoretical improvement of 100×, which is currently reduced by disk and file system overheads.

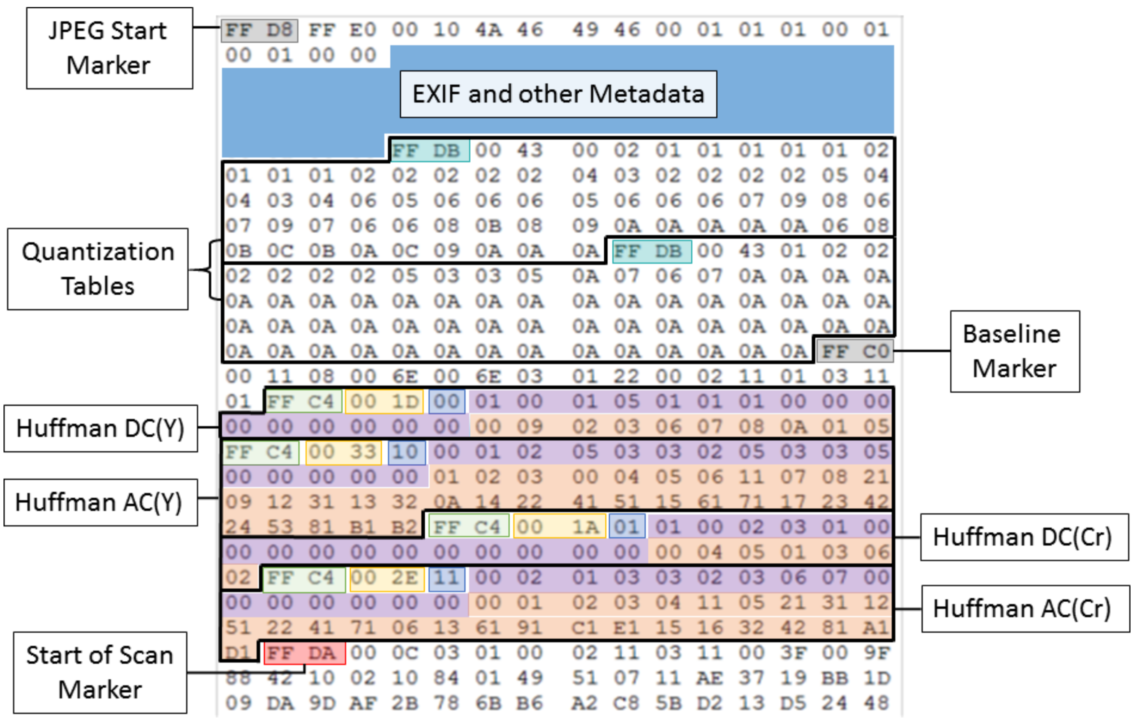
Sub-file signature generation appears to be a promising approach for future forensics, however there is still much work to be done. The following section explores the potential to use a similar sub-file, header based, approach, on images of the JPEG format.

3.5 Sub-file JPEG Signatures Using Optimised Huffman Tables

The JPEG still picture compression standard [77] is the most popular lossy compression system used for images at the time of writing. JPEG images can be found everywhere, from photographs taken by cameras, to thumbnail caches, and images on the Web. JPEG has resisted being replaced by newer, technically superior, schemes such as JPEG 2000 [123], due to its near universal adoption. For this reason JPEG images are incredibly important for contraband based forensics investigations.

This section describes how JPEG compression metadata, found in the file header, may be used to generate highly distinct signatures, while only requiring 1–3% of a relatively small JPEG file to be read. This is achieved by deriving signatures from optimised Huffman tables which are used at the entropy encoding stage in JPEG compression, and is inspired by work in content based image retrieval [23, 108, 24]. While optimised Huffman tables are not used in every image, there has been a recent trend towards optimally encoding images of the JPEG format, such as those published recently by Mozilla [124] and Google [125]. The work carried out in this section has previously been published in the ADFSL Journal (JDFSL) [2].

The remainder of this section begins with an overview of JPEG compression, before describing how these sub-file JPEG signatures are produced in Section 3.5.2. Evaluation of the discriminating power and benchmark performance follow in Section 3.5.3. Supplementary materials, including code snippets, are found in Appendix D.



```
FF D8 FF E0 00 10 4A 46  49 46 00 01 01 01 00 01
00 01 00 00
            FF DB 00 43  00 02 01 01 01 01 01 02
01 01 01 02 02 02 02 02  04 03 02 02 02 02 05 04
04 03 04 06 05 06 06 06  05 06 06 06 07 09 08 06
07 09 07 06 06 08 0B 08  09 0A 0A 0A 0A 0A 06 08
0B 0C 0B 0A 0C 09 0A 0A  0A FF DB 00 43 01 02 02
02 02 02 02 05 03 03 05  0A 07 06 07 0A 0A 0A 0A
0A 0A 0A 0A 0A 0A 0A 0A  0A 0A 0A 0A 0A 0A 0A 0A
0A 0A 0A 0A 0A 0A 0A 0A  0A 0A 0A 0A 0A 0A 0A 0A
0A 0A 0A 0A 0A 0A 0A 0A  0A 0A 0A 0A 0A 0A FF C0
00 11 08 00 6E 00 6E 03  01 22 00 02 11 01 03 11
01 FF C4 00 1D 00 01 00  01 05 01 01 01 00 00 00
00 00 00 00 00 00 00 09  02 03 06 07 08 0A 01 05
FF C4 00 33 10 00 01 02  05 03 03 02 05 03 03 05
00 00 00 00 00 01 02 03  00 04 05 06 11 07 08 21
09 12 31 13 32 0A 14 22  41 51 15 61 71 17 23 42
24 53 81 B1 B2 FF C4 00  1A 01 01 00 02 03 01 00
00 00 00 00 00 00 00 00  00 00 00 04 05 01 03 06
02 FF C4 00 2E 11 00 02  01 03 03 02 03 06 07 00
00 00 00 00 00 00 00 01  02 03 04 11 05 21 31 12
51 22 41 71 06 13 61 91  C1 E1 15 16 32 42 81 A1
D1 FF DA 00 0C 03 01 00  02 11 03 11 00 3F 00 9F
88 42 10 02 10 84 01 49  51 07 11 AE 37 19 BB 1D
09 DA 9D AF 2B 78 6B B6  A2 C8 5B D2 13 D5 24 48
```

Fig. 3.14 The structure of a sample JPEG as it is stored on disk. The metadata section may be long, and it is abbreviated here for clarity.

3.5.1 JPEG Compression Overview

The JPEG standard is a lossy compression technique for reducing the file size of images. The standard leverages properties of human vision in order to provide the best trade-off in perceived image quality to compression ratio, with several stages of compression being utilised.

During compression, images are typically converted to the `YCbCr` colour space, separating the luminance and chrominance channels, the latter of which may be optionally sub-sampled. The result is then divided into 8×8 pixel blocks, which are transformed to the frequency domain using the Discrete Cosine Transform (DCT) to produce a matrix of 64 coefficients. The coefficient at the top left of the matrix, known as the DC (Direct Current), represents the mean colour value of the block, while the remaining 63 coefficients, which are known as AC (Alternating Current), contain horizontal and vertical frequency information. As most of the human sensitive aspects of the signal are concentrated in the top-left of the matrix, which hosts the low frequency coefficients, much of the higher frequency information may be discarded, or represented more coarsely. This is achieved by quantization, with the JPEG quantization table mapping the relative compression ratios of each DCT coefficient. The standard provides a generic quantization table, which can be scaled to vary image quality.

The quantization process results in many AC coefficients becoming zero. A run-length encoding scheme is then applied which compresses these runs of zeroes efficiently.[13] Additionally, as the average colour (DC) of each 8×8 block is expected to change gradually throughout the image, differential coding is used to efficiently compress the colour differences between blocks. The coefficient compression utilises variable length encoding schemes, with the data stored as a set of bit length and value pairs. This information is further encoded using single byte codes, which in turn represent the magnitude of the DC, or combined magnitude and run-length for the AC coefficients. As these codes are repeated frequently, Huffman encoding can be used to compress their representation to variable length bit strings. This allows for frequently occurring codes to be represented in perhaps two or three bits, instead of a byte. The JPEG standard provides a default mapping of these Huffman bit strings for the AC and DC byte codes. However, for more efficient compression, a per image optimised Huffman table may be generated based on the actual occurrences of these codes, resulting in smaller file sizes.

Figure 3.14 depicts the beginning of a sample JPEG image. Immediately following the JPEG start marker are the application markers which specify the particular JPEG form format (such as JFIF, EXIF) and miscellaneous metadata, such as title, comments, camera settings, camera model, or editing software information. This is then followed by decompression information composed of the quantization and Huffman tables. The most basic (baseline) JPEG makes use of two quantization tables, one for luminance, and another for chrominance, with four Huffman tables for the combinations of AC/DC and luminance/chrominance. Both Huffman and quantization tables are mapped to data components through the use of identifiers, which provides some flexibility. Huffman tables are followed by the actual image scan data, which is stored sequentially, from the top of the image to the bottom. An alternative format, the progressive JPEG, may contain multiple image scans, starting with low resolution versions of the image and increasing in steps, allowing for images to increase in quality as they are loaded on the Web. Progressive JPEGs may contain more Huffman tables, which are used for each individual scan.

Mozilla's MozJPEG [124] introduces tweaks to the encoding process by modifying the original JPEG libraries while remaining compliant with the specification. The technique uses optimised Huffman tables. All images are converted to the progressive JPEG format, and new quantization table presets are provided to better accommodate high resolution images. Google's Guetzli [125] takes a more aggressive approach, with coarse quantization presets, Huffman table optimisation and post processing the DCT coefficient matrix. Guetzli produces sequential images, rather than using progressive JPEGs.

[13]This uses a zig-zag pattern from top left to bottom right, as depicted in Appendix Figure D.1

3.5.2 Extracting JPEG Signatures

Unlike with the PNG format, much research has been carried out on JPEG images, for a wide range of purposes. This existing work meant that the process of identifying discriminative elements for signature generation in the JPEG format did not need to start from scratch. Prior work in digital forensics (Section 2.4.1) has shown that compression metadata can be used as a mechanism for detecting modified images and their source software/camera. Content based image retrieval has made use of optimised quantization tables and Huffman tables for the purposes of reverse image searching (Section 2.4.3), which suggests that these optimised tables encode substantial information about the image. Optimised tables are not included with every image (as will be discussed in Section 3.5.3), with optimised quantization tables being particularly problematic to identify due to the wide variety of existing base quantization tables and scalar variants. Taking this into account, optimised Huffman tables were chosen as the sole source of discriminating power in this work. Huffman tables are often the last item included in the header of a JPEG, as depicted in Figure 3.15. This means that, as with the PNG approach in Section 3.4, the image file only needs to be read up to the compressed image scan data. Again, typical values for the offset of the Huffman tables were determined empirically (see Section 3.5.3).

As JPEG is a very popular image format, there were many large datasets available to use. Three datasets were chosen for this work. The first, the Flickr 1 Million dataset [113], contains 1 million JPEGs with optimised Huffman tables. This dataset was chosen because of its large scale and diverse image content, but it also serves as an example of a corpus which is constituted entirely of optimised Huffman JPEGs.[14] The second dataset is the original Govdocs JPEG corpus [114], which contains approximately 109,000 JPEGs. Govdocs was made available largely for forensics research, and contains a wide variety of image encoding parameters and content. This dataset does a better job of representing real world files, as the Flickr corpus is pre-processed by the Flickr hosting platform. The third dataset is a modified version of Govdocs, where all images were optimised using the `jpegtran` [115] utility, with the `-copy all` and `-optimize` flags. Five images failed the optimisation, and were omitted (See Appendix Figure A.1). All images were converted to baseline JPEG during optimisation, with all images in both the Flickr and Govdocs Optimised datasets using the `YCbCr` colour space. The unmodified Govdocs dataset contains mixed colour and types and JPEG types. Duplicate images were not removed from these datasets. File size details are provided in Table 3.1.

[14] With the unexplained exception of three images, 621224.jpg, 636616.jpg, 646672.jpg, which required optimisation.

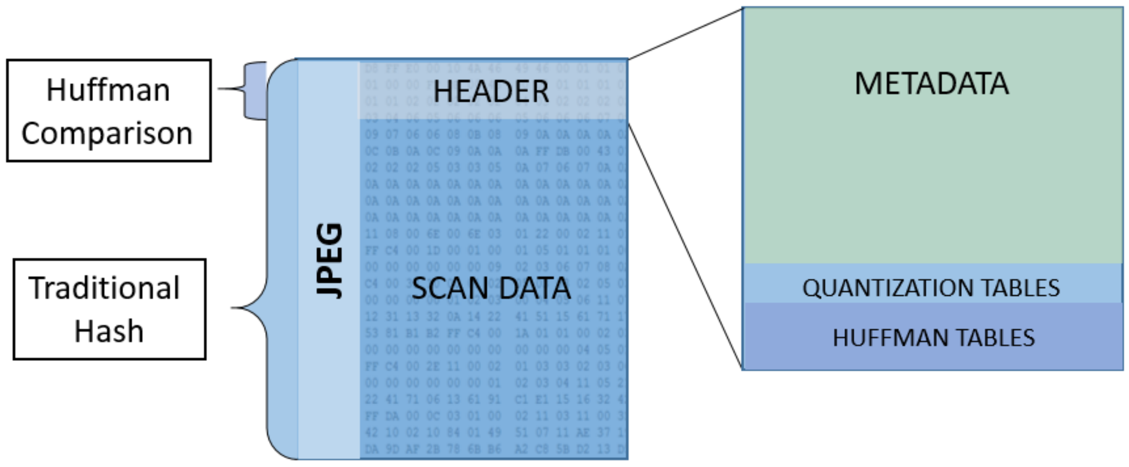


Fig. 3.15 The portions of a JPEG used for traditional cryptographic hashing and Huffman comparison.

Optimised Huffman Tables

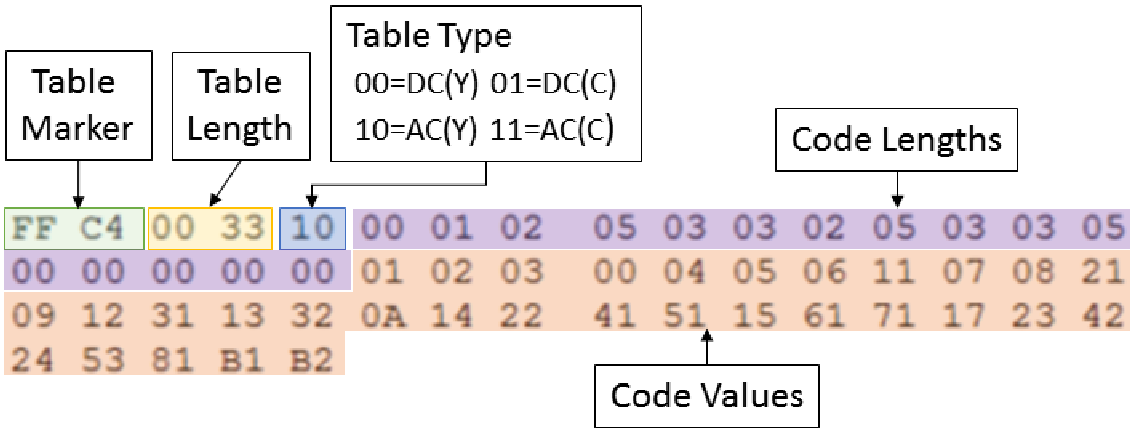
As described in Section 3.5.1, Huffman tables are used in JPEG to entropy encode DCT magnitude/run-order codes, further compressing the data stream. By assigning short bit strings to frequently occurring terms, the overall number of bits required to store the DCT codes can be reduced significantly.

The JPEG standard provides suggested Huffman tables, which were derived from an empirical analysis of a sample of 8-bit JPEGs. However, this is suboptimal as the DCT code distributions will vary based on the content of the image. Optimised Huffman tables, then, are the image specific mappings of these bit strings to the DCT codes, based on their actual frequency of occurrence in that image. The end result is a more efficient compression of the data stream.

While the standard Huffman tables included in the specification have to list mappings for all AC and DC codes, a given image may not make use of all DCT codes. This means that the optimised Huffman table may contain fewer entries, as it only provides those which are required to encode that image. It is therefore not only the mapping of bit strings, but the absence of certain codes, which can be used to discriminate between optimised Huffman tables.

Extracting Huffman Tables

Structures in the JPEG format are preceded by markers which consist of two bytes, which always begin with `0xFF`, with the second byte indicating the type of marker. These markers serve as a roadmap for the decoder, which is necessary as each segment is typically not of a fixed length.



```
FF C4 00 33 10 00 01 02  05 03 03 02 05 03 03 05
00 00 00 00 00 01 02 03  00 04 05 06 11 07 08 21
09 12 31 13 32 0A 14 22  41 51 15 61 71 17 23 42
24 53 81 B1 B2
```

Fig. 3.16 An example Huffman table as it appears on disk. In table types, Y corresponds to the luminance channel, while C corresponds to the chrominance channels (Cb/Cr)

An example Huffman table is provided in Figure 3.16. Huffman tables are essentially stored as two arrays, the first containing the number of Huffman codes of each bit length, while the latter lists the corresponding DC and AC byte codes in the table. These arrays are all that is needed to reconstruct the entire Huffman decode tree. Pseudo-code for generating Huffman signatures is provided in Algorithm 2, showing that an ordered concatenation of all length/value arrays is all that is required to generate a Huffman signature.

In practice, there are many existing libraries for parsing JPEGs. The Libjpeg [117] library was selected to extract the Huffman tables using C++. Once the de-compression object is initialised, all header data up to the start of scan, including the Huffman tables, can be acquired by calling the `jpeg_read_header` function. This header information also includes other useful data for this work, such as JPEG type and colour space. At this point, the file has been processed up to the Start of Scan (SOS) marker, which indicates the beginning of the compressed data streams.

The Huffman signature is then extracted from the `jpeg_decompress_struct` by extracting the arrays for DC and AC tables respectively, ordered by the table type ID. For the purposes of the experiment, matching Huffman signatures were verified using SHA256 digests on the full file to determine if they were true matches.

Properties and Limitations

Optimised Huffman tables provide coarse information about the relative frequencies of DCT codes in the compressed data stream, and therefore communicate some information about the frequency domain representation of the source image. As such, signatures derived from the DCT codes are robust to metadata modification and some small modifications to image content.

Algorithm 2: Generate Huffman Signature. Order tables by type to avoid issues with ordering on disk.

```
Input: JPEG File
Output: JPEG Signature
huffmanTables= {};
marker = nextMarker();
// Loop until SOS marker
while (marker != 0xFFDA) do
    // Check for Huffman marker
    if (marker == 0xFFC4) then
        length = readBytes(2);
        type = readBytes(1);
        // read remaining length after length/type bytes
        htable = readBytes(length-3);
        huffmanTables[type]=toString(htable);
    end
    marker = nextMarker();
end
// Order by table type: 0x00, 0x01, 0x10, 0x11
orderedKeys = order(huffmanTable.keys());
signature = ' ';
for (key in orderedKeys) do
    signature += huffmanTable[key];
end
return signature
```

A limitation of this technique is that both images must have been encoded using the same quantization tables, colour space, and channel sub-sampling. If this is not the case the images will contain a different distribution of DCT byte codes, resulting in different Huffman tables when optimised.

Progressive JPEGs use multiple scans at different resolutions, potentially resulting in many more Huffman tables than baseline JPEGs. As such, encoding the same image as both baseline and progressive JPEG will result in a mismatched number of tables, though the tables from the baseline image may be very similar to the tables for coarse progressive scans. Huffman tables will appear before each scan in progressive JPEG, such that they will be found throughout the file.

The process of generating optimised Huffman tables involves inspecting the corresponding DC and AC streams, and counting DCT code frequencies. The most frequently occurring items are assigned the smallest Huffman codes. The `jpegtran` utility in the Libjpeg package may be used to create optimised images from unoptimised JPEGs with the `-optimize` option. However, this requires the entire file to be processed, and introduces substantial overhead. Therefore, while it is possible to acquire optimised tables for any given image, Huffman table comparison is best suited to images which are already optimised.

Offset Acquisition

As with PNG signatures, the earlier the header ends in the file, the more effective this sub-file approach will be. To explore this, the offset of the Start of Scan (SOS) marker was acquired after reading the JPEG header by calling the `stdio::ftell` function and then subtracting the remaining bytes in the `src` input buffer to get the correct offset.

3.5.3 Evaluation of Discriminating Power

If Huffman tables are to be used to identify particular images in a large dataset, they must contain enough discriminating power to do so. To this end, the Huffman signatures for all images in the Flickr 1 Million dataset were used to construct equivalence classes, grouping together images with the same signature. A class size of *n* indicates that *n* images possess the same signature, with a class size of 1 indicating a unique signature. No alterations were made to the binary or pixel content of images in the datasets, such that reported results are absent of intentional modifications. However, there are instances of very similar image files in the datasets which provide information about how the approach will be affected by intentional modification.

Huffman Distinctness in Flickr 1 Million

Flickr 1 Million	**Equivalence Classes**				
Size	**1 (Unique)**	**2**	**3**	**4**	**5**
No. Images Huffman	999250	726	18	4	5
No. Images SHA256 Duplicates	N/A	722	18	4	5
No. Images Huffman, No SHA Dupes	999250	4	0	0	0

Table 3.9 The number of images belonging to equivalence classes of each size for the Flickr 1 Million dataset. A class size of *n* indicates that there are *n* images with the same signature.

The Flickr 1 Million dataset contains many sets of duplicates, with a total of 746 images having at least one other image in the dataset with identical binary data (see Table 3.9). Once duplicates were removed from the results **all but two pairs of images were found to possess unique Huffman signatures**. This shows that this fast signature generation technique has almost a zero percent false positive rate at scale when searching for identical images.

Indeed, the two sets of JPEGs with matching Huffman tables in this collection are almost identical and in reality differ by a small number of pixels. Image differences were visualised using the Resemble.js library [126], with differences highlighted in Figure 3.17. In the first case, 4 pixels are different as a semi-colon is added to the text rendered in the image, while in the second case one version of the image has two letters transposed. As the matching pairs also use the same quantization tables, such differences are small enough to result in the same optimised Huffman tables being generated.

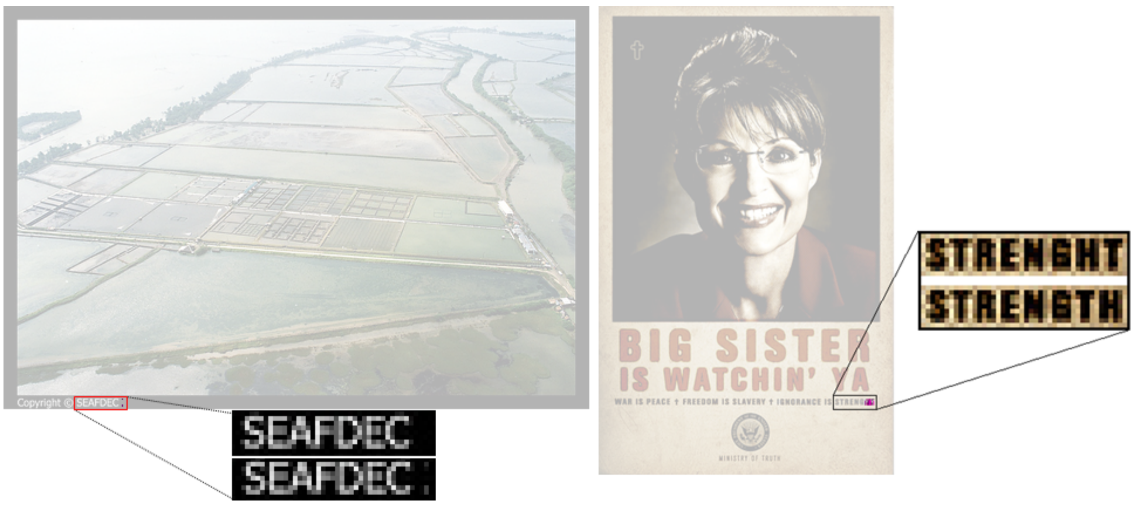


Fig. 3.17 Highlighted image differences for image pairs with matching Huffman tables in the Flickr 1 Million dataset. Images 985964.jpg and 986229.jpg are represented on the left, 431419.jpg and 431931.jpg on the right.

This result shows that optimised Huffman tables possess a great deal of discriminating power, with only two pairs of nearly identical images possessing the same Huffman signature in a dataset of 1 million images. Indeed, this may be seen as a positive property, as the signature can be tolerant to slight changes within the image.

Huffman Distinctness in Govdocs Variants

Colour Space	YCbCr	YCCK	CMYK	RGB	Greyscale
No. Images	95783	3	0	9	13443

Table 3.10 The number of images for each colour space option in the optimised Govdocs corpus.

Two versions of the Govdocs dataset were used: *i)* the unaltered original dataset, and *ii)* a version where all images have had their Huffman tables optimised, and converted to baseline JPEGs, using `jpegtran`. The former is used to derive representative statistics for how common particular types of JPEG are in the wild, while the latter provides a secondary test dataset for optimised Huffman distinctness. It is important to estimate usage of different JPEG configurations in the open world as this will inform the findings of this work. Most datasets are curated and somewhat homogeneous, either in their content, source or post-processing, which means they are not an adequate means of gauging how often particular JPEG features are used. The Govdocs dataset has been acquired from online sources, and while the domains were limited to US government websites, they offer a more accurate, though slightly dated, representation of real world JPEGs than many other datasets. A small number of JPEGs generated errors when optimising or extracting Huffman tables and other information, and those were omitted from this analysis.

The unaltered Govdocs corpus is heterogeneous, with a mix of JPEG modes, colour spaces and origin software. Of approximately 109,000 images, only 6809 JPEGs (6.2%) use the progressive format, with the remainder using the baseline JPEG format. 37,879 (34.7%) baseline JPEGs use default Huffman tables, and, as such, produce the same signature when extracted. 39,035 images (35.7%) contained Adobe application markers, which may either use optimised or pre-defined Huffman tables. The predominant colour space is overwhelmingly `YCbCr`, as shown in Table 3.10.

Prior to optimisation, 39,328 (36%) of the unmodified Govdocs images have a unique Huffman signature, which is 55.1% of images not using default Huffman tables. The remaining images are primarily grouped into very large classes, with 23,706 images belonging to groups of equivalent Huffman tables with 1000 or more members, the largest class containing over 10,000 images. This indicates that, even in the presence of many JPEGs using pre-defined tables, Huffman analysis can be used as the sole method of identifying images more than 1/3rd of the time. That is, this corpus suggests that optimised Huffman tables are used as often as default tables, however the authors argue that the trend is towards optimised JPEGs, and indeed the Govdocs corpus itself is relatively old. The software developed by Mozilla [124] and Google [125] may be an indication that optimised images will appear more frequently on the Web, where page load times and data transfers are relatively expensive compared to other domains.

One caveat is that images taken by the same camera are likely to make use of the same Huffman tables if the camera does not employ image level optimisation. However, with the processing power available on modern devices, and inflating file sizes, optimisation may become more common in the future.

The optimised Govdocs dataset provided similar results to the Flickr 1 Million dataset (see Table 3.11), in that images with matching Huffman tables were either identical in the

Optimised Govdocs	**Equivalence Classes**				
Size	**1 (Unique)**	**2**	**3**	**4**	**5**
No. Images Huffman Only	108539	684	6	0	0
No. Images SHA256 Duplicates	N/A	676	0	0	0
No. Images Huffman, No SHA Dupes	108539	8	6	0	0

Table 3.11 The number of images belonging to equivalence classes of each size for the optimised Govdocs dataset. A class size of *n* indicates that there are *n* images with the same signature.

binary domain, or demonstrate small variations of the same image. In one case, the image content was identical, but a textual description was added in the metadata of one image, changing the SHA256 hash of the file. Both equivalence classes of size three contained variations of the same base image (i.e. all six images were nearly identical, but produced two distance sets of Huffman tables), with very small annotation differences. Another example of almost identical images is provided in Figure 3.18, where an arrow changing place is the only difference in the content.

When combining both datasets into one corpus of over 1.1 million images, no new equivalence classes were found. This confirms that Huffman tables are very distinct when JPEGs are optimised for their content.

3.5.4 Lookup Performance Evaluation

Optimised Huffman tables are distinct at the million image scale, while providing some tolerance for image content modifications. Data reduction is effected by reading only the image header, ideally resulting in faster processing times for optimised images. What follows in this section is an analysis of the potential performance increases of this approach on datasets with relatively small files.

Typical Start of Scan Marker Offsets

Statistics for the position of the Start of Scan marker are depicted in Table 3.12 for all three JPEG datasets, with a visual representation for Flickr and unmodified Govdocs in Figure 3.19. As the SOS marker appears after the Huffman tables, the data shows that very few 4096 byte media blocks are required to read those Huffman tables. In the case of the Flickr dataset, a single block read suffices for 96.6% of the dataset, with three blocks being sufficient for 99.6% of images. The figures are slightly higher for the Govdocs corpus, which contains more metadata, where nine blocks are required to acquire 99% of

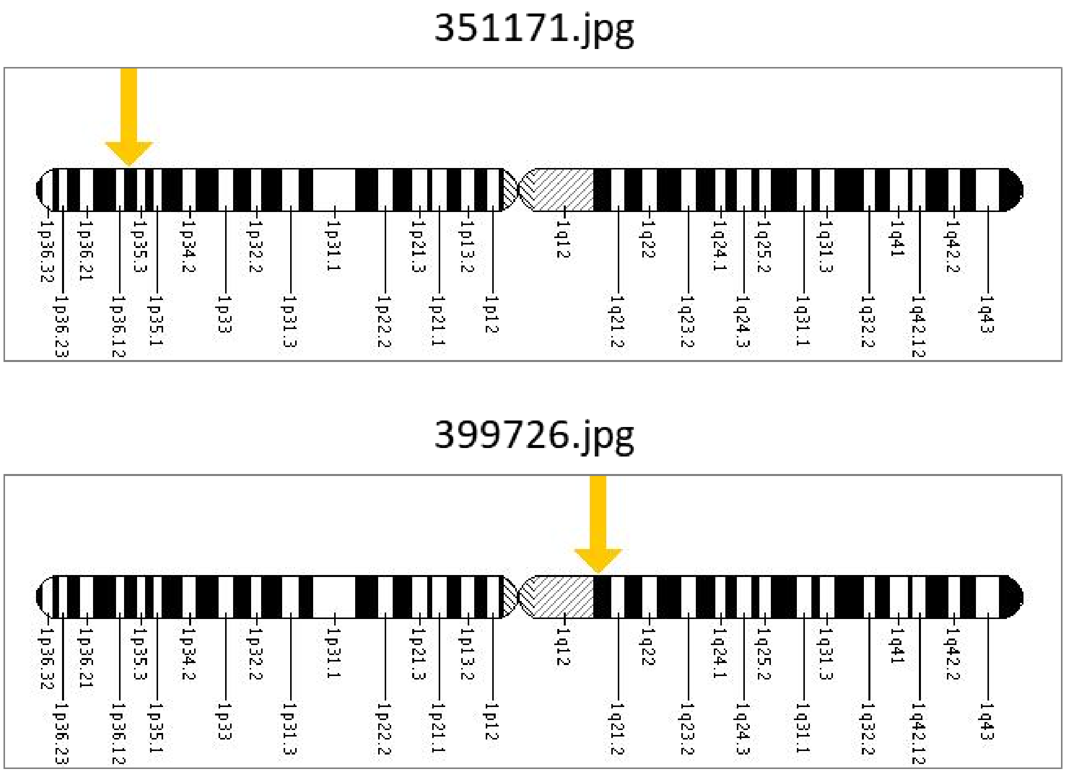


Fig. 3.18 Example matching Huffman tables from the optimised Govdocs dataset. Borders around images added for emphasis. The only difference between the images is the placement of the arrow.

Huffman tables. In both cases, the distribution is long tailed, with the majority of images requiring a single block.

Using mean values for marker offsets and file lengths, 1.6% of the file must be read on average to acquire the SOS marker in the Flickr 1 Million dataset, while both Govdocs datasets require 1.2% of the file to be processed. However, when considering that 4096 bytes may be the minimum transfer size on modern storage media, the figure for the Flickr dataset rises to 3.2%, while Govdocs remains all but unchanged. Using Huffman based sub-file signatures, a small fraction of the file is all that is required to be read, as opposed to the entire file for traditional hashing.

Number of Codes and Table Lengths

The maximum number of DCT byte codes possible in the baseline JPEG format is 348 (12 per DC, 162 per AC table). However, the maximum number of codes observed for an optimised JPEG in this work was 277, suggesting that the number of codes may be used as a heuristic to distinguish between optimised and pre-defined tables. However, as can be seen in Figure 3.20, not all pre-defined tables use all codes. The spikes in the bottom

Dataset	**Percentile** (B)					**Mean** (B)
	50	75	95	99	99.9	
Flickr 1 Million	973	3560	3599	9060	28866	2054
Govdocs	623	4181	23926	36128	51658	4205
Govdocs Optimised	417	3972	23863	36205	51426	4080

Table 3.12 Start of Scan offsets in bytes for all datasets.

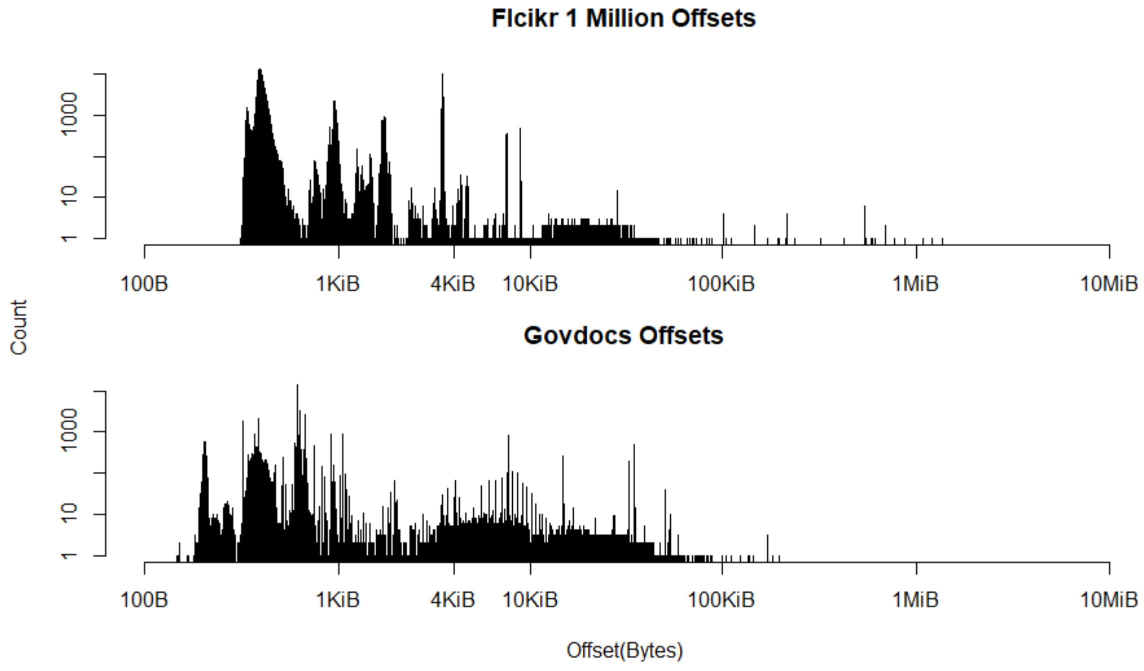


Fig. 3.19 Log-log distribution of Start of Scan offsets for Flickr 1 Million and unmodified Govdocs. Optimised Govdocs is almost identical to the original, and is therefore omitted.

graph of Figure 3.20, at 174 and 249 codes, are caused by images produced by Adobe Photoshop's 'save for web' settings, which optimise entropy encoding using alternative mechanisms. However, based on this data, images with less than 300 codes are very likely to make use of optimised JPEGs.

Ignoring Huffman table markers, the length of the Huffman table may be calculated by summing the number of entries in the value and length vectors for each table. The maximum possible number of codes is 348 for baseline JPEGs, plus 16 bytes of marker and metadata for each of the four Huffman tables (64 bytes), for a total of 412 bytes.[15]

[15]However, if the baseline JPEG uses greyscale colour, it only requires a single DC and AC table (luminance only), for a total of 206 bytes.

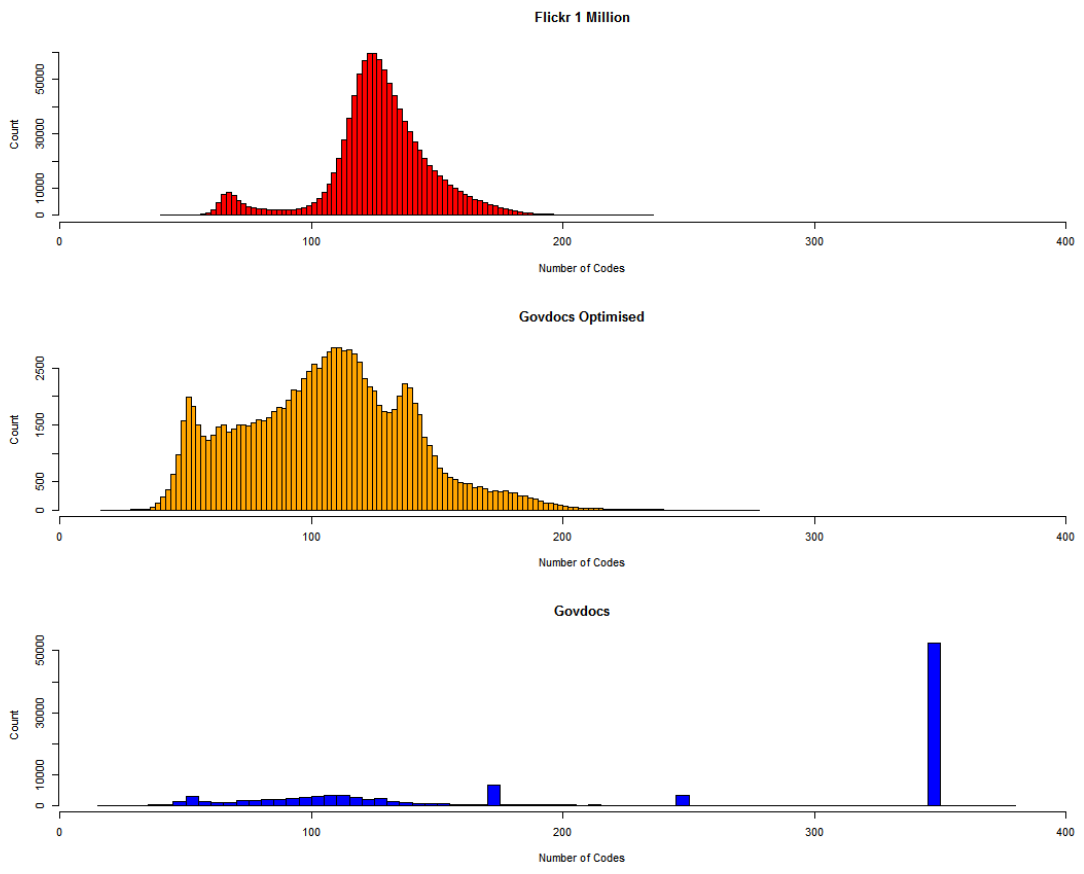


Fig. 3.20 Distributions for the number of DCT codes in each dataset. Flikr 1 Million the top, Optimised Govdocs in the middle, and unmodified Govdocs at the bottom.

The maximum length found for an optimised JPEG in this work is 300 bytes, with a mean of 191 bytes for the Flickr dataset.

Using this observation, it is possible to identify images with a high degree of certainty which use unoptimised tables. Additionally, this table length information indicates the number of bytes which are required to be stored for JPEG Huffman signatures. To reduce storage, these signatures themselves could be hashed using a cryptographic hashing mechanism with fixed length digests.

3.5.5 Timed Benchmarks

Benchmarks compare the Huffman sub-file method against traditional full file hashing using the SHA256 algorithm. Benchmark times correspond to the duration for extracting signatures from a list of files, without storing the signatures or performing database lookups. The time to enumerate files on the disk is also not included in the total. As PNG benchmarks explored the performance characteristics of accessing files in different orderings, only the normal file order provided by Python's `os.listdir` function was used in these benchmarks. In order to assess the IO costs of accessing small pieces of the file from the storage media, an additional benchmark was performed which read the first 4KiB of each file without any processing, to act as a minimal disk baseline.

The same two computers as the PNG benchmarks in Section 3.4.4 were used to evaluate performance: *i)* a workstation - i5-4690k, 16GiB DDR3 RAM, Western Digital Red 4TB HDD, Crucial MX300 525GB SSD, and *ii)* laptop - i7-5500U, 8GiB DDR3 RAM, Samsung 840 EVO 500GB SSD (OS). Storage benchmarks are provided in Appendix B.1. Again, several multi-threading options were used. Testing was limited to the Flickr 1 Million dataset, as this is both the largest dataset, and the worst case performance scenario (with the mean file size being 1/3 that of Govdocs, such that the header is a larger proportion of the file). Benchmarks were carried out on Ubuntu 15.04 64bit, with memory caches being cleared between runs.[16] Benchmarks were run automatically in sequence using a script, similar to those in Appendix C.1. As caches were cleared between runs, and background processing controlled for, there should be no bias between iterations.A C++ application was compiled in g++ (version 4.8.4) using Boost 1.55 for thread pools, libjpeg62 for JPEG parsing, and OpenSSL for cryptographic hashing (SHA256). Files were copied to the test drives sequentially, with no fragmentation, in OS order. Volumes were then mounted read only for benchmarking.

A summary of timed benchmark results and improvement factors is depicted in Figure 3.21, with a breakdown of more detailed results thereafter.

HDD Performance

The Huffman based sub-file method saw no improvement when analysing Flickr 1 Million on the HDD. This can be attributed to the small file sizes of this dataset, as discussed in Section 3.3.1. These file sizes are perhaps less than 1/8th of the size we can expect from a relatively modern camera. As the small block (random 4KiB) read performance of mechanical media is very low, this leads to processing which is no faster than sequentially reading the entire file in this dataset. However, performance gains can be expected on larger files, as suggested by the PNG benchmarks on slightly larger files in Section 3.4.4.

Another key property of sub-file approaches is that they appear to have mixed results when making use of highly threaded approaches on hard disk drives. Adding small numbers of threads can either slightly improve or degrade performance, while high thread counts of 16+ generate too many non-sequential requests for efficient HDD access. This is also the case for the full-file hashing approach, as well as the sub-file PNG approach (see Figure 3.8). However this is not surprising, as the most efficient method for reading a

[16]Again using the command: `sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'`

		Mean Time (s)				
Flickr 1 Million	Threads	Sub-file Huffman	First 4KiB	Fullhash	Sub-File Improvement	
Workstation NTFS HDD	1	1100.0	1099.7	1374.3	1.2	
	2	1104.3	1096.0	1123.0	1.0	
	4	1122.7	1122.7	1097.7	1.0	
	8	1236.0	1252.4	1173.7	0.9	
	16	1701.3	1546.0	1303.0	0.8	mean:
	32	2176.0		1476.7	0.7	0.9
Workstation NTFS SSD	1	341.3	283.3	866.3	2.5	
	2	290.3	271.0	605.7	2.1	
	4	282.0	267.0	639.7	2.3	
	8	278.3	267.0	692.7	2.5	
	16	279.3	267.3	704.7	2.5	mean:
	32	278.7	270.3	699.7	2.5	2.4
Laptop NTFS SSD	1	291.8	219.5	950.0	3.3	
	2	227.5	196.3	714.7	3.1	
	4	203.3	189.5	569.0	2.8	
	8	203.8	191.0	604.3	3.0	
	16	204.0	192.3	606.3	3.0	mean:
	32	203.8	191.8	593.7	2.9	3.0
Workstation EXT4 SSD	1	282.0	246.0	1046.5	3.7	
	2	149.0	135.0	602.3	4.0	
	4	89.0	79.3	376.8	4.2	
	8	59.8	53.8	285.3	4.8	
	16	49.3	45.8	269.0	5.5	mean:
	32	47.0	44.0	269.0	5.7	4.7

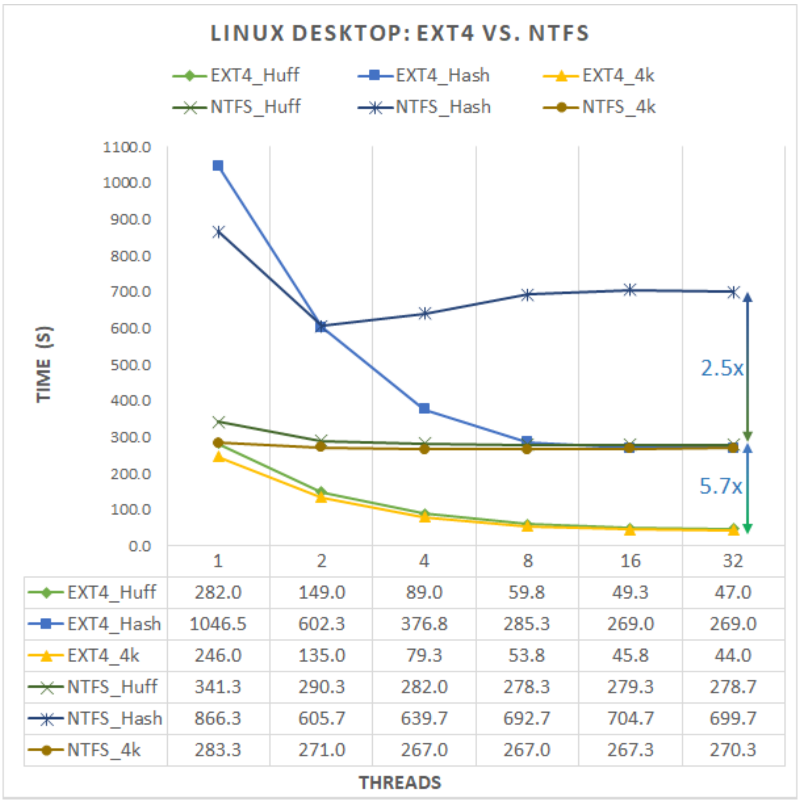
Fig. 3.21 Summary of benchmark results for the Sub-file Huffman approach, Full File Hash (Fullhash), and First 4KiB file reads for different configurations. Normal file ordering was used in all cases.

hard disk is a single sequential pass, as noted in the literature on digital forensics triage (Section 2.2).

SSD Performance: NTFS and EXT4 (Linux)

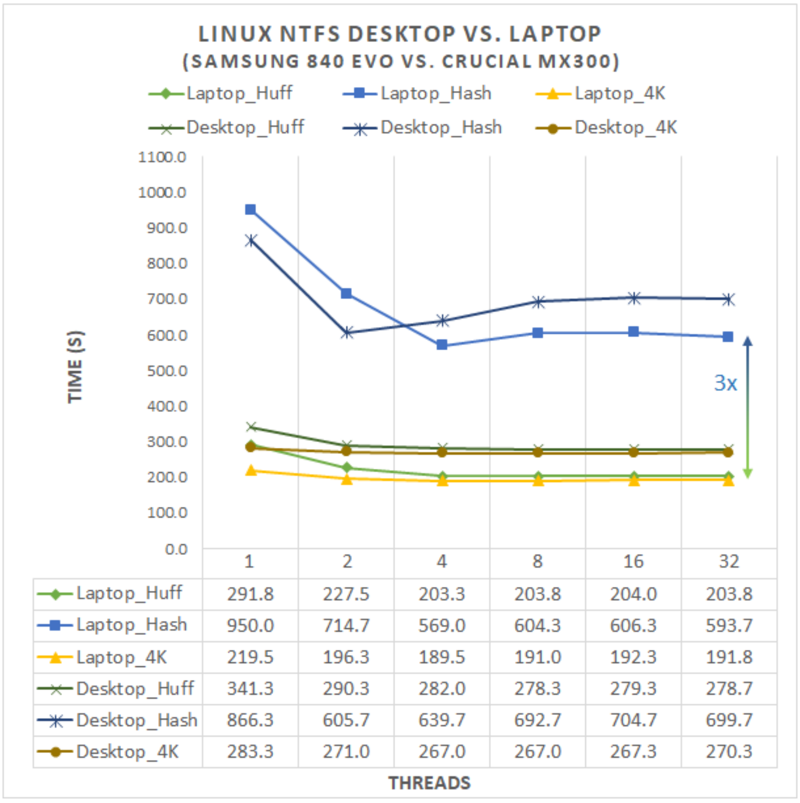
Once again, substantial performance gains were seen when utilising solid state media, which are better suited to random access patterns. Figure 3.22 show results for Huffman signature extraction and full file hashing. Results are compared across both the EXT4 and NTFS file systems on Linux to better control for background processing and keep the experiment consistent. However, as previously noted in the PNG experiments, the `ntfs-3g` driver introduces overheads, which allows for the exploration of the impact of slower file systems on the sub-file approach, even if it is artificial in this case.

Both file systems scale well to two threads, however NTFS performance appears to plateau at this stage, while the EXT4 file system performance improves even up to 32 threads. When comparing Huffman signature extraction to full file hashing on the



	1	2	4	8	16	32
EXT4_Huff	282.0	149.0	89.0	59.8	49.3	47.0
EXT4_Hash	1046.5	602.3	376.8	285.3	269.0	269.0
EXT4_4k	246.0	135.0	79.3	53.8	45.8	44.0
NTFS_Huff	341.3	290.3	282.0	278.3	279.3	278.7
NTFS_Hash	866.3	605.7	639.7	692.7	704.7	699.7
NTFS_4k	283.3	271.0	267.0	267.0	267.3	270.3

Fig. 3.22 Comparison of the relative performance of Huffman and Hash signature extraction to reading the first 4K file block, across the EXT4 and NTFS file systems. EXT4 performance is close to using raw LBA block addresses.



	1	2	4	8	16	32
Laptop_Huff	291.8	227.5	203.3	203.8	204.0	203.8
Laptop_Hash	950.0	714.7	569.0	604.3	606.3	593.7
Laptop_4K	219.5	196.3	189.5	191.0	192.3	191.8
Desktop_Huff	341.3	290.3	282.0	278.3	279.3	278.7
Desktop_Hash	866.3	605.7	639.7	692.7	704.7	699.7
Desktop_4K	283.3	271.0	267.0	267.0	267.3	270.3

Fig. 3.23 Comparison of the relative performance of Huffman and Hash signature extraction to reading the first 4K file block, across two computers. The Laptop SSD possesses better random 4K Read performance and higher IOPS.

workstation, a speed increase of 2.5× was achieved on NTFS, and up to 5.7× on EXT4. EXT4 producing better overall performance is consistent with the results of benchmarking the PNG sub-file approach, as depicted in Figure 3.12.

As both the JPEG Huffman and PNG sub-file approaches only require a single 4KiB disk block for most files, their performance should be roughly equivalent. However, the mean file size of Flickr 1 Million is approximately 1/10th of the size of the Govdocs PNG dataset, with the median file being roughly 1/3rd of the size. Despite this, reasonable performance gains are still possible with the sub-file approach on SSDs. As the file size decreases, more emphasis is placed on disk and file system overheads, which appear to make a significant difference for both small and large files alike.

To explore the best-case performance of the sub-file Huffman approach, the First 4KiB of each file was read, without further processing. This allowed the Huffman marker parsing and infrequently large file headers, to be ruled out as a bottleneck. This data is also included in Figure 3.22. For both file systems, Huffman extraction performance mirrored 4KiB file read performance very closely, typically with less than 10% additional overhead. This suggests that Huffman signature extraction is close to the theoretic limits of fractional file access via the file system, with additional costs taking the form of additional block reads when the header stretches more than a single 4KiB disk block.

Disk and File System Overheads

The relatively poor performance of NTFS could be attributed to overhead within the file system, which does not scale well with many concurrent accesses. As this experiment was conducted on Linux, a verification run was completed on Windows to rule out the Linux `ntfs-3g` driver as a bottleneck. On Windows, better performance was achieved on NTFS for both sub-file hashing and full file hashing, particularly when the thread count was higher than two. However, these results are accurate for the benchmarks conducted on the Linux platform, and the PNG benchmarks in Section 3.4.4 show a performance difference in favour of EXT4, despite running the NTFS benchmarks on Windows 10.

File system overheads were explored by obtaining the logical block addresses (LBA) for each file and running the experiment with physical addresses, rather than looking up each file in the file system. When the initial pre-processing was not included in the recorded time, benchmark times using the LBA addresses performed nearly identically to those of EXT4. This suggests that EXT4 has minimal overhead for file access when compared with NTFS on Linux, but also that pre-processing the NTFS Master File Table ($MFT) can alleviate driver performance overheads, at the cost of upfront initial processing.

When a storage device with higher random 4KiB read performance is used, the relative performance of Huffman extraction to full file hashing also improves. This is depicted in

Figure 3.23, which compares the same signature generation techniques and 4KiB reads across both the workstation and laptop machines. The laptop SSD has higher small block throughput with low queue depths (40.74 MB/s on Laptop vs. 28.60 MB/s on the workstation), which is of benefit to the sub-file approach despite the less performant CPU. This difference is particularly evident when comparing the Huffman and 4KiB read performance in Figure 3.23, with 37% improved performance on the laptop, for an improvement of 3× over full file hashing.

Partial file access performance with HDDs are dominated by seek time, with transfer time being less of a factor. However as file sizes increase the relative performance would be expected to improve, both for mechanical media and solid state devices. SSDs have much smaller effective seek times, and thus the proposed technique holds significantly more appeal for flash media. In modern systems flash media are becoming increasingly common, particularly in the laptop arena, in addition to already dominating the mobile device market. The observation that partial file access scales well on this type of media opens the door for flash storage optimised approaches to digital forensics, which may well be the dominant storage technology for personal computers in the near future.

3.5.6 Confirmation Hashing

Device	**Threads**	**Sub-file Time (s)**	**Hit Rate (% Contraband)**			**Fullhash Time (s)**
			1%	**10%**	**33%**	
Workstation NTFS HDD	8	1227.0	1266.9 (3.1×)	1626.0 (2.5×)	2555.8 (1.6×)	3990.3
Workstation NTFS SSD	8	278.3	285.3 (2.4×)	347.6 (2.0×)	590.0 (1.4×)	692.7
Laptop NTFS SSD	8	203.8	209.8 (2.9×)	264.2 (2.3×)	405.0 (1.5×)	604.3
Workstation EXT4 SSD	8	59.8	62.6 (4.6×)	88.3 (3.2×)	154.7 (1.8×)	285.3

Table 3.13 Fullhash confirmation impact on the Huffman sub-file approach. 1%, 10% and 33% of corpus assumed to generate a positive hit, respectively. Calculated by adding the appropriate percentage of Fullhash time to the Sub-file approach, and therefore ignores the benefits of caching. Relative improvement factors in brackets.

From the evaluation of discriminating power in Section 3.5.3, optimised Huffman tables were shown to be highly discriminative, and essentially unique at the million image scale. In cases where images produced the Huffman table, in every case it was because images were almost identical, such that no image was a true false positive. This provides greater discriminating power than the PNG signatures in Section 3.4, which would produce roughly 1900 false positives at the million image scale on a homogeneous dataset.

The results of this experiment suggest that it is not necessary to perform follow-up confirmation hashing for Huffman signatures, as they are highly accurate and also allow for minor image modifications. However, figures are provided in Table 3.13 for this scenario.

On a dataset of such small files, confirming hashes on 1/3rd of the dataset drops the benefit of Huffman signatures to 40–80%. However, as the Huffman based approach has similar characteristics to the sub-file PNG approach, the relative benefit of this approach should increase with file size, even when performing confirmation hashing.

3.5.7 Outcome

The main contribution of this section is an inexpensive method for creating signatures of JPEG files. Huffman tables are present in the header of every JPEG, with the signature generation method in this work exploiting Huffman tables which have been optimised for maximal compression, which is an increasingly common encoding technique used on the Web. An analysis of the distinctness of such signatures is provided, as well as an examination of the portion of the file which must be processed to extract them. Finally, the method is evaluated with timed benchmarks comparing the extraction process to a traditional hashing method.

Similar to the PNG sub-file signatures in Section 3.4, Huffman based signatures can be acquired in most cases by reading a single 4KiB disk block, requiring 1–3% of the file even on the small file sizes of the Flickr 1 Million dataset. These signatures only produced matching Huffman tables for nearly identical images at the million image scale, and as such provide a robust mechanism for identifying contraband images.

While the small file sizes rendered no performance increase on the HDD, SSD performance was 2.5–3$\times$ that of full file hashing on NTFS, reaching 5.7$\times$ on EXT4, despite the very small files sizes. As this method is again bound to random 4KiB disk performance and file system lookup performance, it is expected that the approach will perform better on future storage technologies. Additionally, increasing file sizes will see a corresponding performance increase in-line with the PNG sub-file approach, as they both process approximately the same amount of data with little CPU overhead.

3.6 Sub-file Signatures in Practice

Sub-file signatures appear to be capable of high levels of discrimination and performance, even on very small file sizes. The remainder of this chapter further discusses the implications and characteristics of file type specific sub-file signatures.

3.6.1 Extending to Additional File Types

Both sub-file approaches in this chapter are used to show that file specific sub-file approaches, which utilise the structure of that file type, have potential. It was not feasible to develop an approach for all file types, however, there are essentially two main strategies which arose from the work in this chapter.

The first, used for the PNG signatures, is to extract discriminating power from features of the file type which are always, or commonly, present. Ideally the modification of these features should result in the content of the file being changed in some way, either at the rendering, or compression, stage. This provides a degree of robustness to arbitrary, content-preserving, metadata manipulation. Using this approach, even if the file format does not contain an appropriate number of high entropy, highly discriminating features, a reliable signature may still be extracted from the combination of these common pieces of information.

The second approach, used for the JPEG Huffman signatures, is to identify a potential high entropy source of information within the file type, which is somehow related to the content of the file. Ideally, this should be something which can be construed as a coarse representation of the file content. Optimised Huffman tables serve this role, as they are calculated based on image statistics. Other sources of such high entropy data include low resolution scans of the image (such as in progressive JPEG), compression dictionaries and colour maps. If a file type possesses one of these high entropy sources in the file header, then it may be possible to use it as a single discriminating feature for signature generation. This approach also has the benefit of being somewhat sensitive to content manipulation in the body of the file, rather than the header.

When extending the sub-file approach to other file types, the process should ideally start with the latter signature generation technique, making use of coarse content representations. Relating the signature to the content of the file in some way provides a level of assurance which is not necessarily implied by a combination of low entropy features. However, as most media files of forensic interest will be compressed in some way, the likelihood of there being a significant number of compression domain features to use is reasonably high. When applied to larger media, such as video files, the amount of data to be read can increase significantly, while still remaining a small percentage of a typical file. In this case, both header data and a number of data blocks may be used to generate the signature, while still providing substantial performance gains. For example, features may be derived from early keyframes in the video, or the motion vectors included in the MPEG specification [127]. This thesis was scoped to focus on image forensics, and as such an approach for video files, or other MIME types[17] such as audio, was not developed. It is likely that an appropriate approach for larger media files may have to look at encoding information throughout the entire file, processing it as a stream, rather than simply looking at the file header. The reason for this is that as the file gets larger, the header and early data becomes less representative of the overall content of the file, which may result in lower accuracy for files which are not binary duplicates. This would require a slightly different approach to that taken in this thesis.

While optimised Huffman tables are currently not used in every modern JPEG, the move to optimise images in this way is an indicator that JPEG compression is no longer sufficient. The standard is decades old and lacks support for modern features, such as high bit depth, HDR, and alpha channels. JPEG compression also results in larger file sizes than necessary, with blocky artefacts and reduced fidelity. Indeed, the Apple iPhone switched to a newer image standard, High Efficiency Image Format (HEIF) [128], to combat the growing footprint of high resolution images on mobile devices. The compression employed by these new standards offers an opportunity to extract signatures from features which are key components of the file type. For instance, JPEG 2000 [123] makes use of multiple resolutions, which can be used to extract thumbnail style representations from the beginning of the file. Other formats, such as HEIF, make use of intra-coding techniques originally developed for video codecs, relying heavily on prediction. In these cases, as with PNG, there may not be a representation of the full content at the beginning of the file, but low cost signatures may be extracted from prediction features, though this may be more useful for semantic hashing schemes than data reduction approaches.

A final concern is the mechanism for comparing signatures. To prevent computational complexity, features comparison should ideally be based on hashable feature vectors or strings, as this allows for O(1) lookups in a hash table. If the similarity of features is instead required, then solutions such as bloom filter comparisons may be appropriate, as with the hash comparisons made by *sdhash* [38].

3.6.2 Discriminating Power and File Features

The level of discrimination achieved by a sub-file signature derived from file format features cannot be generalised from this work, and is entirely dependant on what elements

[17]https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

are present in a given file type. The approach taken for PNGs shows that a handful of features may be combined to produce highly discriminative signatures (Figure 3.7), and that small portions of scan data are also a powerful feature (Figure 3.6). However, the discrimination performance for PNG is not sufficient for standalone deployment on very large image sets, and would generate 20,000 false positives for every 10 million images. The cost of increasing the uniqueness of the signature here, is to read more scan data, or to derive features from non-critical parts of the file, which are not related to the content of the image. This would then reduce the robustness of the approach to arbitrary metadata manipulations which do not change how the image is rendered. If the use case is simply a faster method for detecting binary level duplicates, then the PNG approach can likely be extended for greater accuracy by reading more pixel scan data, or including additional metadata features. However, if arbitrary, content preserving, metadata changes can trivially break such approaches, perhaps the forensics community should aim higher, targeting a minimum level of robustness against small-scale, arbitrary, manipulations.

The work on JPEG demonstrates that certain compression metadata may be enough to uniquely identify a given image in a dataset of millions of images (Figure 3.9), while also providing resistance to EXIF metadata modification, or small content manipulations (Figure 3.17). However, optimised Huffman tables are not the default for every JPEG and cannot be used without pre-processing on many existing images. When focusing purely on obtaining fast results, this additional processing may negate the benefit of the approach, as the entire JPEG would have to be read to generate the optimised table, at which point traditional cryptographic hashes could be used for the same IO cost. However, optimised Huffman tables are still useful when the IO overhead is less of a concern and an inexpensive signature is required which is tolerant of small scale manipulations. Unfortunately, as this is a feature derived from image statistics, it will not detect images which use different encoding parameters, such as a different quality level (quantization mask scaling), as the DC codes generated for the Huffman table will have different distributions. In this sense, the approach is brittle to a particular kind of content-preserving manipulation.

Ultimately, signatures generated using either approach on other file types will have to make trade-offs in terms of discriminative power, performance, and robustness. If the criteria for such signatures can be formalised, it is likely that a conforming sub-file signature can be generated for most file types. The hierarchical processing deployed in spam detection is another reason to be optimistic about this general approach. If the signature is only intended to filter images, rather than identify them, then an accuracy of 90% is viable, as long as there is a substantial overall performance benefit when verification stages are taken into account.

3.6.3 Data Reduction and Storage Technologies

Theoretically a data reduction of 100-fold should net a corresponding performance increase. However, this is never going to be the case in practice, both due to limitations of physical processing, and overheads introduced by storage media and file systems. This was evident for benchmarks in both the PNG and JPEG approaches, where the relative performance benefit was much greater on solid state media. The minimum file access overheads are significant, and reduce the benefits of accessing a tiny fraction of file data. This may be described by a cost model for each file transaction:

$$TransactionCost = Index\_Overhead + Storage\_Overhead + Data\_Transfer\_Cost \tag{3.1}$$

For large files, the cost of transferring an entire file should overwhelm the cost of looking it up in the file system (index), as well as any storage overheads associated with locating the disk blocks in question, and moving read heads, if applicable. As the effective file size decreases, or less data is read from the file, the cost of transferring the data is proportionally less, such that it may be overwhelmed by the overheads. Indeed, this coarse model reflects the behaviour of the sub-file benchmarks in this chapter. The relative sub-file performance increases with file size, as the full file hashing transfer cost becomes very high, while slower storage media and file system overheads hinder the performance of sub-file approaches.

Storage_Overhead is a generalisation of many low level storage media behaviours, but it can be captured well enough by the random 4KiB read performance of a device. This is because sub-file techniques most commonly only require a single 4KiB disk block, and match the performance of reading the first 4KiB of each file closely. As random 4KiB performance increases relative to sequential throughput, the relative benefit of reading fractional files increases. This is evident from both the differences between HDD and SSD performance, and the improved relative performance of the laptop SSD over the workstation SSD.

Table 3.14 depicts the effective throughput of the sub-file and fullhash methods across each of the benchmarked machines and datasets. The effective sub-file throughput for NTFS on SSDs is lower than the disk benchmarks with a request queue depth of one, achieving roughly 6% of the maximal 4KiB throughput for the device. EXT4 SSD benchmarks for the workstation show a significant improvement when the file system overhead is reduced, achieving 25–40% of the maximum 4KiB throughput. Full file hashing also suffers a penalty on NTFS, achieving 40–60% of the maximum sequential throughput on the device, as opposed to 85% on EXT4. It is also clear that file size

Maximal Effective Read Speed (MiB/s)					
Device	Approach	Effective	Raw Disk Benchmark		Effective % of Max
			Q1	Q32	
Workstation NTFS HDD	Sub-file JPEG Sub-file PNG	3.6 0.8	0.56	1.9	189.5% 42.1%
	Fullhash JPEG Fullhash PNG	110.2 91.9	128	128	86.1% 71.8%
Workstation SSD NTFS	Sub-file JPEG Sub-file PNG	14.5 13.8	38.8	226.6	6.4% 6.1%
	Fullhash JPEG Fullhash PNG	199.8 416.6	496.9	524.6	38.1% 79.4%
Workstation SSD EXT4	Sub-file JPEG Sub-file PNG	87.1 58.3	38.8	226.6	38.4% 25.7%
	Fullhash JPEG Fullhash PNG	448.9 457.2	496.9	524.6	85.6% 87.2%
Laptop SSD NTFS	Sub-file JPEG Sub-file PNG	20.1 15.6	27.3	254	7.9% 6.1%
	Fullhash JPEG Fullhash PNG	212.7 329.3	412.6	509.4	41.8% 64.6%

Table 3.14 The effective read speeds of the benchmarked approaches compared with device benchmarks. The sub-file total data is estimated at 4KiB per file, while fullhash uses the entire size of the dataset. Q1 and Q32 represent device request queue depths of 1 and 32, respectively.

impacts effective file system overheads, with the larger PNG files resulting in much higher sequential throughput than for JPEGs. On the hard disk, the cost of the file system is much less pronounced due to the slow mechanical read heads. Additionally, larger file size is actually detrimental here to relative throughput performance, possibly because a single buffered read on the mechanical media can capture more than a single file when they are very small.

With these observations it is possible to speculate about how these sub-file techniques would perform on more advanced storage technologies in the future, given that they provide a performance benefit over full file hashing even when using a small fraction of the theoretical 4KiB throughput. By comparing the ratio of random 4KiB (queue depth 1 (Q1)) performance to sequential throughput (queue depth 32 (Q32)), it should be possible to determine a worst-case performance scenario for the sub-file approach. On the workstation SSD, Q1 4KiB throughput is approximately 7% of the maximal sequential throughput. Benchmarks of more recent SSDs using the M.2 connector and the NVMe protocol [129] show roughly a 5% for the sustained 4KiB throughput as a percentage of contiguous sequential read speeds.[18] In this case, we may expect sub-file techniques on the NVMe device to perform slightly worse, as it seems the ratio is slightly less favourable. However, the NVMe protocol was designed as a NAND flash replacement for the SATA protocol, and as a result, has fewer overheads. This increase in efficiency may effectively reduce the Storage_Overhead, and result in a net performance gain, despite the worse ratio. A completely different scenario is given by a drive which makes use of the Intel Optane technology[19] [129]. In this case, the random 4KiB throughput is 31% of the maximum sequential throughput, suggesting that sub-file performance on this device could overwhelmingly outperform full file hashing at any file size.

The benchmark in this work makes use of small files, but also operates on unfragmented, sequential files. We can therefore expect full file hashing to be a strong baseline, as contiguous data benefits full file hashing, and small files are to the relative detriment of sub-file approaches. Given that sub-file methods perform better than this baseline, with this low overall disk utilisation, file system overhead, it is likely that real world performance will outshine the benchmarks in this chapter. Indeed, the fragmented sequential throughput of NAND SSDs is significantly lower than for contigous data, though the Intel Optane technology appears unaffected [129].

There are many factors which affect the performance of the sub-file technique, with nuances which are not captured by Equation 3.1. However, it is clear that the focus of future storage technologies is not magnetic media, and such devices may offer a better performance trade-off than current technologies. Additionally, file system overheads may be reduced in the future as operating systems and file systems optimise for non-mechanical media.

3.6.4 Applicability to Networked Device Contraband Detection

Both of the PNG and JPEG sub-file signatures discussed in this chapter process a stream of data looking for particular markers. When the markers are found, only a small amount of data needs to be collected before the signature can be generated. As such, both techniques may be used to detect contraband on a network data stream, where the data can be observed until the appropriate file markers are discovered. If there is too much data to be processed,

[18]Samsung 960 Pro 512GB, 114 MiB/s sustained 4KiB, vs. 2226 MiB/s sustained sequential.

[19]ADATA XPG SX8200 240GB, 661 MiB/s and 2125 MiB/s for sustained 4KiB and sequential throughput, respectively.)

the stream can be sampled for the appropriate markers, triggering the collection of the data stream for a brief period. Assuming packets order can be reconstructed, it should be simple to extract the necessary features for signature creation This process avoids the relatively expensive process of hashing data blocks from the entire stream, with effective data reduction around two orders of magnitude.

When detecting contraband on a remote device [130], this approach allows for a small subset of the data to be sent across the network, in cases where forensics processing cannot be carried out directly on the remote device. However, as these sub-file approaches are stream based, this will involve many small data requests/seeks across the network when traversing the file. This is particularly problematic when carrying out an investigation across the Internet, as large packet round trip times introduce a substantial overhead. This scenario is explored in more detail for the generic sub-file approach in Section 4.5.

3.6.5 Digital Forensics Processes

Sub-file techniques offer significant speed improvements when processing a device for contraband. In a digital investigation, this reduction in processing time would be best utilised in the triage stage of an investigation, where fast results are critical (see discussion in Section 2.2). Sub-file approaches operate at the logical file level, rather than the physical disk level, and therefore requires the file system to be mounted. This means that random disk block sampling approaches, working at the physical level, are not directly compatible with sub-file methods. However, the file system level data reduction used by Grier and Richard [43] would be a good counterpart for this approach, for two reasons: *i)* Data subsetting in Grier and Richard is achieved by applying heuristics after parsing file system metadata. This results in a set of logical files, which can then be immediately processed using the appropriate sub-file approach. *ii)* Since the file system metadata is parsed in order to subset files on the disk, LBA addresses for files can be extracted for little additional cost while the heuristics are applied. This would allow files to be accessed without excessive file system overheads, and can achieve performance akin to EXT4 on NTFS partitions, regardless of operating system.[20] While the sub-file approaches described in this chapter could process all relevant files on a disk, this would mean that many innocuous files would also be processed, such as application icons and other images built into the operating system or mundane applications. By applying this approach to the subset of data most likely to be relevant, both approaches can work in tandem to generate rapid results. The combination of techniques also has the benefit of being applicable to a live system, write blocked disk, or forensic image.

[20]Based on benchmarks carried out for Section 3.5.4

Whether used for triage, or to speed up a full forensic analysis on a disk image, confirmation hashing may be employed if the sub-file signature is not deemed accurate enough. In this case it would be recommended to carry out confirmation hashes, or other verification steps, after all relevant files have been processed. This allows for the best compromise between fast results and robustness, allowing the analyst to quickly get an idea of the state of contraband on the device. Alternatively, experiments could be conducted to optimise the time to process both sub-file hashes and full file hashes on files which show positive hits. This could exploit caching technology on disk, as the entire file does not need to be re-read, or some of the file may already be in the read buffer. The sub-file portion may also be stored in memory along with a map of the LBA addresses, such that redundant reads are not performed for the full file hashing stage.

3.7 Conclusions

This chapter explored the possibility of creating robust, and accurate, sub-file signatures for the forensic detection of contraband media. Two general approaches were taken, with an implementation for the two most popular image formats on the Web.

The first approach, applied to PNGs, combines a number of low entropy header features in the file to create a signatures. These signatures were shown to be 99.8% accurate on a worst-case homogeneous dataset, with false positives being generated by solid background colours. Even on a dataset of relatively small files, significant speed improvements were found over traditional full file hashing, particularly on SSDs and the EXT4 file system. The second approach, applied to the JPEG format, uses coarse content representations, again found in the file header, to generate signatures which were essentially unique at the million image scale. An analysis of false positives showed that they were actually modified versions of the same file, and would therefore still relevant to an investigation. This approach reads roughly the same amount of data, with similar performance characteristics to the PNG approach.

Both techniques were evaluated against the contraband signature criteria from Section 3.2 throughout the chapter, and have proven to be fit for the task. A particular property of these approaches is that they are bound to small block disk performance, and are therefore better suited to NAND flash storage media. This work effectively lays the foundation for future forensics techniques which take advantage of the properties of modern non-mechanical media, which may be the key to dealing with forensic backlogs in the future.

Chapter 4

Generic Sub-file Signatures

4.1 Introduction

The sub-file techniques in Chapter 3 are file type specific, and require an in-depth knowledge of all relevant file types to generate discriminative signatures. In contrast, the work in this chapter sets out to create sub-file signatures which are file type independent, which should work equally well on all files without knowledge of their structure. Small pieces of the file are hashed in lieu of the entire file, with the approach being somewhat similar to block-based hashing techniques in the literature [29, 44–46]. The work presented here differs in that the goal is to reduce the amount of data to read from disk, with hashes being performed at the logical file level, instead of the physical disk level.

Rather than focus on a stream-based approach, which allows for non-essential file information to be skipped, or treated appropriately, the approach taken in this chapter is block-based, and simply samples the file from a particular offset. In practice, this can be achieved by sampling arbitrary blocks in the file, but for the purposes of this work only the beginning and end of the file was considered. Despite being file type agnostic, datasets in this chapter contain both JPEG and PNG files, such that the discriminatory power and generalisability of the technique can be evaluated, as well as providing a variety of file sizes to work with.

Block-based sub-file generation alleviates some of the overheads and difficulties of the file-specific approach, as it should generalise to all file types, including those yet to be adopted. Ideally, as few blocks as necessary should be read from the disk to effect the same kind of data reduction achieved in Chapter 3, and attain the associated speed benefits. The work in this chapter is also evaluated in terms of the criteria identified in Section 3.2.

Dataset	**File Type**	**No. Files**	**Mean Size**		**Median Size**	
			KiB	**MiB**	**KiB**	**MiB**
Flickr 1 Million	JPEG	1000000	124 KiB	0.12 MiB	117 KiB	0.11 MiB
Govdocs PNG	PNG	108885	1426 KiB	1.39 MiB	344 KiB	0.34 MiB
Flickr Subset	JPEG	25000	118 KiB	0.12 MiB	112 KiB	0.11 MiB
Flickr Subset PNG	PNG	25000	295 KiB	0.29 MiB	295 KiB	0.29 MiB
Govdocs Sub. PNG	PNG	25000	535 KiB	0.52 MiB	152 KiB	0.15 MiB

Table 4.1 Details of the datasets used in this chapter.
The full Flickr 1 Million and Govdocs PNG datasets are used for the local disk experiments in Sections 4.3 and 4.4, while 25,000 image subsets were used for the network storage experiments in Section 4.5.

4.2 Description of Datasets

The local disk experiments in Sections 4.3 and 4.4 make use of the full Flickr 1 Million and Govdocs PNG datasets discussed in the previous chapter. Section 4.5 explores the performance of generic sub-file approaches in networked environments, which have much lower throughput than local storage devices. As such, Section 4.5 makes use of 25,000 image subsets of the Flickr 1 Million and Govdocs PNG datasets in order to make experiment run time manageable, and to save on cloud storage costs. Details of the datasets are provided in Table 4.1. No modifications in the binary or pixel domain were made to the images, with the exception of converting Flickr images to PNG for one of the subsets. Subsets were chosen to create three distinct datasets with increasing file sizes to demonstrate file size scaling performance of sub-file approaches in networked environments.

The first subset, Flickr Subset, is composed of the first 25,000 Flickr 1 Million images in numerical file order (0.jpg, 1.jpg, 2.jpg .. 24999.jpg), and is 2.81 GiB total. No modification was made to these files. The second subset, Flickr Subset PNG, is the same 25,000 images converted to PNG to increase their file size, totalling 7.04 GiB total. Once again, the Python Pillow library [116] was used for conversion to the PNG format. The final collection is the first 25,000 images of the Govdocs PNG dataset, as listed by Python's `os.listdir` function, and is the largest subset at 12.7 GiB. Further details are provided in Appendix A.2. The same file size considerations apply from Section 3.3.1, as all datasets possess a median file size under 350 KiB, with the mean file size of all subsets under 550 KiB.

4.3 Generating block-based Sub-File Signatures

Signature generation in Chapter 3 essentially focuses on identifying potential sources of entropy in a given file type. In contrast, the approach taken here is to develop strategies for hashing small pieces of a file to generate a unique hash digest for that file. This technique was previously touched on during the analysis of PNG small block hashes in Section 3.4.3. What follows is a more in-depth examination of a variety of sub-file hashing strategies. The work in this Section, and Section 4.4, has been published at the IEEE Cyber Security 2018 Conference [3]. Supplementary materials are found in Appendix E.

4.3.1 Sub-file Hashing Strategies

Three distinct sub-file approaches were used in this work, with the generalised form containing a parameter, *n*, indicating the number of bytes constituting a read block. This data is then hashed using the SHA256 algorithm to produce a file signature.

First n: Read *n* bytes from the **beginning** of the file. This was exemplified by 4KiB and 80KiB for *n* in this section.. 4KiB was chosen as it represents the fastest possible sub-file hash, corresponding to the smallest read unit on contemporary storage drives for both solid-state and mechanical media. It is also a reasonable approximation for the performance of prior sub-file signature generation schemes in Chapter 3. The higher boundary of 80KiB (20×4KiB data blocks) corresponds to approximately $2/3$ of the mean file size of images in the Flickr 1 Million dataset, and is chosen to represent the largest chunk of a file which could still be considered to be 'sub-file' for most images. These read blocks are always expected to be byte aligned with modern hard disk sectors and SSD pages, which are 4096 bytes.

Last n: Read *n* bytes from the **end** of the file, exemplified by 4KiB and 12KiB for *n* in this section. The former value is chosen to maximise speed, while the latter was chosen to highlight the performance trade-off when the size is increased slightly by one or more disk blocks. These read blocks are not expected to align with disk sectors or SSD pages often ($1/4096$ of the time), and in practice two or four sectors/pages, respectively, would be accessed by the underlying storage media.

First n+Last n: Read *n* bytes from **both** the start and the end of the file, resulting in a total of $2n$ bytes being read from disk. This technique is used both as a method to potentially improve discriminative power, but also to highlight block retrieval performance characteristics of non-contiguous data blocks. *n* is 4KiB in this work, and was chosen to represent the most effective performance/discrimination trade-off. This results in 8KiB of data to hash, and often three disk sectors/SSD pages to be accessed by the storage media.[21] This strategy is abbreviated to First+Last *n* for the remainder of this thesis.

Prior work on block-based hashing [29, 44–46] operates at the physical level, which requires that file-disk block alignment be accounted for. This is important as different devices may use different block sizes, with 512 bytes being the traditional hard drive sector size, though hard drives since 2011 have transitioned to the *Advanced Format*, which makes use of 4KiB (4096 byte)[22]sectors (with legacy 512-byte block emulation) [131]. The block hashing strategies described above operate at the logical file level, and therefore it is not important if hashed data falls on particular disk block boundaries, except when considering the performance characteristics of the approach. While the First *n* approach should be block aligned, and therefore generate the same block hashes as prior work, Last *n* is not expected to be block aligned, and produces different block hashes. The difference in what data is hashed is depicted in Figure 4.1.

It should also be noted that the robustness of the above sub-file hashing strategies and prior block-based hashes differs. In prior work, adding a single byte to a file would shift the alignment of data in all blocks, changing all subsequent block hashes for the file. In contrast, the sub-file strategies are only affected when the hashed data is modified in some way, as absolute disk alignment is ignored. Additionally, slack data at the end of the disk block which is not marked as part of the file in the file system will also not be included in the sub-file hash, though arbitrary data which is appended logically will still be included. This means that while non-essential data may be considered in the sub-file hash, the signature is still more robust than traditional full file and block-based hashing approaches. By using cryptographic hash digests directly, signatures are also fixed length, and can take advantage of constant time, O(1), lookups.

Avoiding Reading from the Middle of a File - Offset shifting

An important omission above is that none of the strategies sample from an arbitrary point in the middle of the file - where middle is defined as some point which does not overlap with the beginning or end of the file. Whether picking an offset within the logical file, or on a physical disk, it is trivial to modify the content of the offset by prepending data prior to the offset, shifting all of the data along by however many bytes were added. This means that simply adding in some EXIF data to an image file, or adding a single null byte somewhere in the header, would mean that strategies based on sampling from a particular

[21]Always one block for First 4KiB, and one or two blocks for Last 4KiB

[22]Some non-consumer SAS drives may make use of wider "4kn" sector sizes, which includes 4112, 4160, and 4224 byte sectors. These larger sectors will provide additional space for error correcting code, but do not appear to be used in consumer hardware.

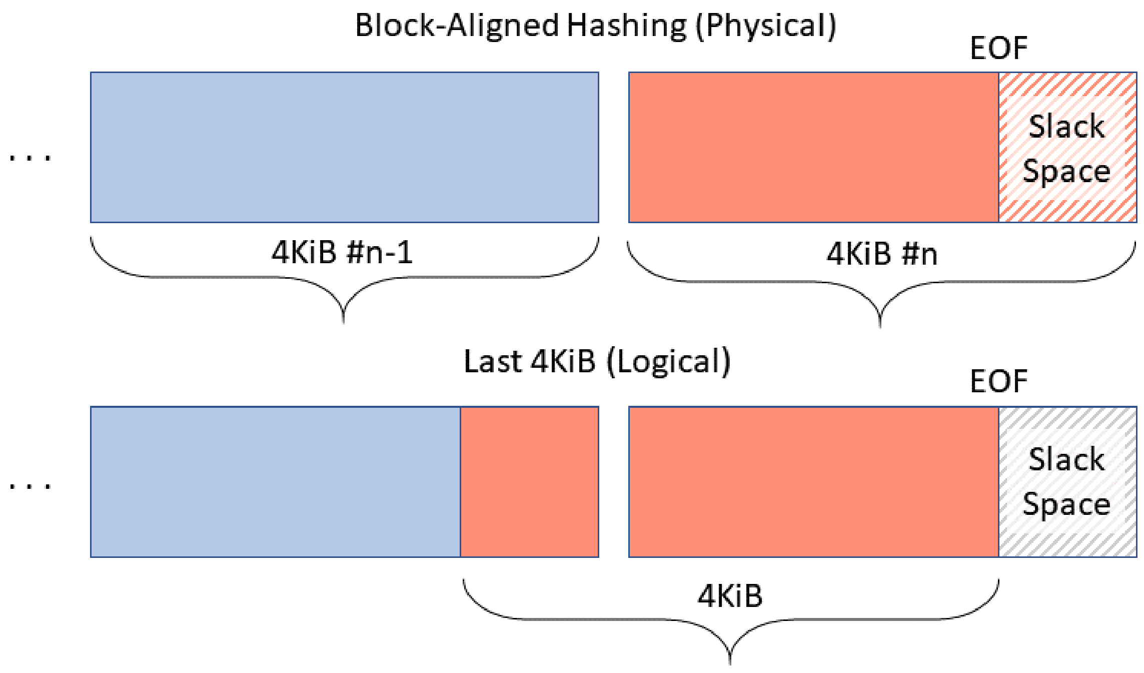


Fig. 4.1 Hashed data comparison of sub-file Last 4KiB and traditional block hashing. Each outer rectangle represents a disk block (HDD sector, SSD page). The last 4KiB hashes the last logical 4KiB of file data, while physical level block hashing ignores the End of File (EOF) and hashes the entire physical block, including any slack space.

offset would be defeated. As the goal of the above sub-file approaches is to be fast, but also mitigate small, arbitrary changes, this would be unacceptable.

The Last *n* approach avoids the offset problem by calculating the offset to be read from the end of the file, such that adding data prior to this point will have no effect on the hashed data. While this approach could be used for hashing locations in the middle of a file, these offsets could be shifted in the other direction by adding arbitrary data after the proposed offset, rather than before it, resulting in the same problem. Of course, the Last *n* strategy is not immune to this problem, as adding data to the very end of the file will shift the intended data, however this means that the appended information has to be in a very specific place, making it both detectable and arguably less likely.

4.3.2 Evaluation of Discriminating Power

A fundamental property of a file signature is that it is unique when used for file identification, or almost unique when employed for filtering schemes. The higher the degree of discriminating power, the fewer false positives which either have to be verified by automated or manual means. Table 4.2 shows the number of unique signatures for each sub-file hashing strategy on each dataset, not including full file hash duplicates (SHA256).

Number of Unique Signatures				
Dataset	**First 4KiB**	**First 80KiB**	**First+Last 4KiB**	**Last 4KiB**
Flickr 1 Million (no Fullhash duplicates)	970633 (97.10%)	999349 (99.97%)	999622 (100%)	999622 (100%)
Govdocs PNG	108670 (99.80%)	108824 (99.94%)	108885 (100%)	108885 (100%)

Table 4.2 The number of unique signatures for various sub-file hashing strategies across both datasets. First 80KiB is included to show that even reading a substantial part of the beginning of the file does not produce unique values. Duplicate files, as determined by full file SHA256 digests, are omitted.

Hashing 4KiB of data from the beginning of the file generates discriminative signatures, however, it does not produce unique signatures for either dataset. Indeed, this method previously proved less effective than hashing 4KiB of compressed pixel data in Section 3.4.3. Reaching farther into the file, 80KiB of data still does not provide a unique signature, though it gets close, with over 99.9% unique digests on both datasets. The problem appears to be twofold: *i)* Files with large amounts of non-distinct metadata, such as repeated colour profiles and camera metadata, can cause many images to contain the same, or very similar header files. As headers may extend hundreds or thousands of kibibytes, this means that some small block hashes inevitably produce the same digest.[23] *ii)* As noted for images of the PNG format (Section 3.4.3), solid background or transparency can produce the exact same signature if the discriminating pixel content is not reached by the hashed block.

For these reasons, the First *n* strategy is not effective for producing unique signatures, though it may be adequate in some environments where homogeneous file headers are not a concern. Instead, this approach should be seen as a performance baseline when using a block size of 4KiB, as it was employed during the overhead evaluations in the previous chapter.

In contrast, reading as few as 4KiB from the end of the file generated unique signatures for both datasets, precluding the need to process larger blocks from the end of the file. The explanation for the difference in performance between reading from the start and end of the file can be found in the nature of compressed file structures. Compressed files typically put metadata first, followed by the compressed data stream, which may be

[23]As Huffman tables typically appear at the end of the header, this is still possible even with optimised Huffman tables.

terminated by an end of file marker (as with JPEG's `0xFFD9`). The last 4KiB, then, should be almost entirely composed of compressed file data. As the very nature of compression is to increase information density, compressed data streams are typically very high entropy. This high entropy stream is very unlikely to be produced by unrelated images, resulting in a highly distinct signature.

While the performance evaluation below, in Section 4.4, sets a minimum of 4KiB for the value of *n*, far fewer bytes can be used to generate discriminative signatures in these datasets. The Last 16 bytes of each file in the Govdocs PNG dataset generated unique signatures for all but a single pair of images[24], while 32 bytes produces unique signatures for the entire dataset. Unique values for the Flickr dataset were produced when reading between 1024 and 2048 bytes from the end of the file. This means that, even for one million images, less than 4KiB of data is required to produce a unique signature at this scale. However, the remainder of this chapter continues with a minimum size of 4KiB, setting more than the bare minimum level of accuracy, at the expense of a slight drop in performance.[25]

The First+Last 4KiB strategy also generates unique signatures as it contains the Last 4KiB, with the possibility that it may scale better to tens, or hundreds, of millions of files. For the purposes of performance analysis, the First+Last *n* technique is included both to show the effective performance of the technique, but also as a means of highlighting the behavioural characteristics of retrieving multiple, non-contiguous, data blocks from a file.

As a final sanity check, the Last 4KiB approach was tested on the University of Southern California's Videotake dataset [132], which contains 1924 videos, with a mixture of MP4 and MOV formats. Some files failed to download at the time, resulting in a subset of 1866 images being tested. All videos which were not full file hash duplicates produced a unique Last 4KiB signature, which, while limited, shows that the discriminatory power is not limited to the JPEG and PNG formats.

4.4 Performance on Local Storage Media

In order to evaluate the potential processing speed improvements of each sub-file hashing strategy, a direct comparison is made with traditional full file hashing using the SHA256 algorithm. Benchmarks were carried out on the same workstation from the evaluations in Chapter 3 (i5-4690k, 16GiB DDR3 RAM, Western Digital Red 4TB HDD, Crucial MX300 525GB SSD). As before, neither drive hosted the Operating System, and both the NTFS

[24] 897530.png and 919147.png

[25] Reading 2KiB of each file would result in a single disk block being read 1/2 of the time, while a 4KiB block will only align with the block boundary 1/4096 of the time, therefore requiring two blocks 4095/4096 of the time.

and EXT4 file systems were evaluated. The code was written in Python, and executed in the Python 2.7.12 interpreter on Ubuntu 16.04 LTS. SHA256 hashes were calculated using Python's `hashlib` library, which is implemented in C. Datasets were copied to the drives sequentially, and file orders were determined by Python's `os.listdir` function. Volumes were mounted with the `-ro,noatime` flags to prevent modification (see Appendix E.1.1 for full list of flags). In copying files to an empty drive sequentially, and using their ordering provided by the file system, sequential read performance is maximised. This ensures a strong baseline performance from full file hashing. Multiple thread counts were used, experiments were repeated three times and memory caches were cleared between runs.[26] Reported times do not include file enumeration times from `os.listdir`. Code snippets are provided in Appendix E.1.

4.4.1 Timed Benchmarks

Sub-file hashing strategies must be appreciably faster than full file hashing to be useful, and should be effective across different media types and file sizes. Each hashing strategy was applied to both the Flickr 1 Million and Govdocs PNG datasets. Performance metrics are provided in Table 4.3 for the Flickr 1 Million dataset, and Table 4.4 for Govdocs PNG. The left-hand column for each technique indicates the mean total time (seconds) for processing the dataset, while the right-hand side contains relative performance factors to full file hashing. The Fullhash technique has no performance factor, as it would simply be compared to itself.

The First 80KiB strategy was omitted as it has little to gain over reading the First 4KiB of the file, while 12KiB of data from the end of the file was tested to determine the impact of additional small data blocks on the most promising technique. Hard disk data is provided only for NTFS runs with a single thread, as multiple threads decrease performance in some cases, and sequential access is critical to maximal throughput on hard disks.

Several factors affecting the performance of techniques in Chapter 3 are used to aid in the evaluation of these sub-file hashing strategies, below.

Drive Type

Shadowing the results of Chapter 3, the drive type, and its small block, random 4KiB, read performance are the most important performance factors. Disk overhead properties are essentially the same whether reading from the start or end of a file. On the HDD, these overheads overwhelm the data transfer costs (previously discussed in Section 3.6.3), such

[26] Again using the command: `sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'`

Sub-file Benchmarks: Flickr 1 Million

Mean hash Time (s) and Relative Performance Factor to Fullhash

FS / Drive	**Threads**	**First 4KiB**		**First+Last 4KiB**		**Last 4KiB**		**Last 12KiB**		**Full-hash**
	1	307.3	2.6	703.3	1.2	481.3	1.7	571.8	1.4	813.0
	2	287.1	2.4	619.5	1.1	408.5	1.7	491.4	1.4	678.3
NTFS	4	275.7	2.5	542.6	1.3	347.5	2.0	448.7	1.5	690.8
SSD	8	270.2	2.7	488.7	1.5	305.5	2.4	405.7	1.8	724.1
	16	270.4	2.7	493.2	1.5	309.2	2.4	410.7	1.8	735.0
	32	271.6	2.8	524.9	1.5	319.7	2.4	411.6	1.9	766.7
	1	257.8	3.4	502.7	1.8	282.0	3.1	393.4	2.2	882.8
	2	139.3	4.1	282.0	2.0	153.2	3.7	225.6	2.5	565.0
EXT4	4	82.7	4.5	169.2	2.2	90.8	4.1	137.6	2.7	370.3
SSD	8	55.6	4.9	114.3	2.4	59.0	4.6	88.9	3.1	273.9
	16	45.6	5.8	89.5	2.9	42.7	6.1	63.3	4.1	262.1
	32	44.3	5.9	81.2	3.2	37.9	6.8	59.8	4.3	259.1
NTFS HDD	1	1593.2	0.8	1587.0	0.8	1605.6	0.8	1621.3	0.8	1284.9

Table 4.3 Generic sub-file benchmark results for the Flickr 1 Million Dataset. Mean time to read and hash (SHA256) on the left, colour coded performance factors relative to Fullhash on the right.

Sub-file Benchmarks: Govdocs PNG

Mean hash Time (s) and Relative Performance Factor to Fullhash

FS / Drive	**Threads**	**First 4KiB**		**First+Last 4KiB**		**Last 4KiB**		**Last 12KiB**		**Full-hash**
	1	30.7	25.1	94.0	8.2	48.5	15.9	55.8	13.8	770.3
	2	27.9	21.4	74.4	8.0	39.1	15.2	47.6	12.5	595.3
NTFS	4	27.4	22.3	57.6	10.6	36.6	16.7	43.6	14.0	609.8
SSD	8	26.9	22.1	49.2	12.1	31.4	18.9	40.0	14.8	593.4
	16	26.8	22.1	50.0	11.9	32.5	18.3	40.2	14.8	593.8
	32	27.7	22.0	53.0	11.5	33.6	18.1	40.8	14.9	609.0
	1	28.8	25.7	53.6	13.8	31.0	23.8	43.0	17.2	739.3
	2	15.3	32.7	30.0	16.6	17.0	29.4	24.4	20.5	500.1
EXT4	4	9.0	39.7	18.0	19.8	10.0	35.7	14.9	24.0	357.8
SSD	8	6.0	54.2	12.0	26.8	6.5	49.9	9.5	33.9	323.1
	16	4.8	65.9	9.2	34.3	4.6	68.6	6.6	47.6	315.2
	32	4.7	66.6	8.0	38.7	4.4	70.6	5.8	53.4	310.5
NTFS HDD	1	691.6	2.6	818.9	2.2	665.9	2.7	673.6	2.6	1776.8

Table 4.4 Generic sub-file benchmark results for the Govdocs PNG Dataset. Mean time to read and hash (SHA256) on the left, colour coded performance factors relative to Fullhash on the right.

that it makes little difference if the block size is 4KiB or 12KiB, or where the blocks are located in the file. No benefit on the hard disk was provided over full file hashing by any of the sub-file hashing techniques on the Flickr 1 Million dataset, while Govdocs PNG performance was between $2.2\times$ and $2.7\times$ faster.

As expected, the sub-file strategies perform much better on the SSD relative to full file hashing. The different strategies also separate in performance more clearly on this kind of media. Reading the First 4KiB of the file is the fastest approach overall, with Last 4KiB shadowing it closely, only suffering around a 10–15% performance overhead. In some cases Last 4KiB is slightly faster, however this may simply be due to how blocks were organised on the EXT4 partition, which allowed for slightly higher parallel access to the end blocks of files by chance.[27]

First+Last 4KiB falls considerably behind the other approaches, and is about twice as slow as Last 4KiB in the worst cases. In spite of this, First+Last still manages a small performance increase over full file hashing on the small Flickr files, with solid benefits on the larger PNGs. Reading an extra two blocks from disk with Last 12KiB incurs a penalty of around 20–35% over Last 4KiB. This suggests that most of the difference between First 4KiB and Last 4KiB is due to the extra disk blocks which are accessed by misaligned logical data at the end of the file.

Overall, the behaviour of the Last 4KiB approach closely mirrors the performance of the sub-file approaches in the previous chapter, but generally appears to be slightly faster overall. This is likely due to the block-based approach, which reads the blocks as a unit, rather than processing a stream, which necessitates slight delays between acquiring blocks as file markers are parsed.

File Size

File sizes also have a very large impact on performance, which is particularly clear now that the same techniques are being run on datasets with substantially different file sizes. The mean file in the Govdocs PNG dataset is roughly $11\times$ larger than in Flickr 1 Million, which is reflected in the large difference in sub-file performance on both datasets. This can make the distinction between being effective on hard disk drives, or being slower than simply reading the entire dataset sequentially (given an unfragmented dataset). The difference is also very pronounced on SSDs, with the relative benefit of Last 4KiB over full file hashing reaching $6–7\times$ on Flickr, and an impressive $70\times$ on Govdocs PNG. The gulf in performance is of the same order of magnitude as the size difference of the datasets.

[27]SSDs are composed of multiple NAND modules which can be accessed in parallel by the flash controller. Larger SSDs are typically slightly faster than smaller drives, as they have more modules to access in parallel. If certain blocks are more evenly spread across the NAND modules, then they will be faster to access in parallel.

This is very encouraging, as even the Govdocs PNG dataset contains relatively small files when compared to high-resolution camera images. The relationship between the different hashing strategies remains the same as file size increases, with Last 12KiB falling between Last 12KiB and First+Last 4KiB.

The performance scaling of sub-file techniques with file size is of particular importance, and is explored in more detail in Section 4.4.2.

File System and Overheads

The sub-file techniques in both the current and the previous chapter have shown EXT4 to outperform NTFS at higher thread counts, particularly on Linux, which appears to have a less performant NTFS driver implementation. The idea of file system overheads was previously explored in Section 3.6.3, However, the benchmarks in Table 4.3 and Table 4.4 facilitate further insight. The total amount of data in the Flickr dataset is 118 GiB, with 148 GiB in the Govdocs PNG dataset, spread across one million images and 108,885 images, respectively. The total time taken to fully hash all files in the Govdocs PNG dataset takes slightly longer than the Flickr dataset on both file systems. This means that the cost of hashing an extra 30 GiB of data in the PNG dataset overwhelms the cost of accessing 908,885 additional files. The opposite is true for sub-file hashing strategies. As the minimum overhead of accessing a file is the primary limitation of sub-file approaches, ten times as many files results in ten times the processing time, regardless of the total amount of data in the dataset.

As these benchmarks were conducted on Linux, processing is about twice as fast on EXT4 than NTFS, however this gap should close substantially when processing NTFS partitions on Windows. Though, as previously discussed, parsing the $MFT for LBA addresses can overcome this limitation on Linux. Despite the driver overhead, the NTFS performance of Last 4KiB is still significantly faster than full file hashing in all cases.

4.4.2 Performance Scaling With File Size

As file size has such a large influence on the performance of sub-file hashing strategies, it was explored in more detail by acquiring per-file benchmark estimates. In order to control for the potential variance in measuring small IO operations, files were grouped into size bins of width 8192 bytes[28], discarding bins with fewer than eight files. The average times for full file hashing and Last 4KiB were then calculated from each bin on the SSD with the EXT4 file system. Figure 4.2 depicts a scatter plot of mean per-file times for each bin.

[28]i.e. the first bin contains all files of size 1 byte to 8192 bytes, the next bin contains files from 8193 bytes to 16384, and so on.

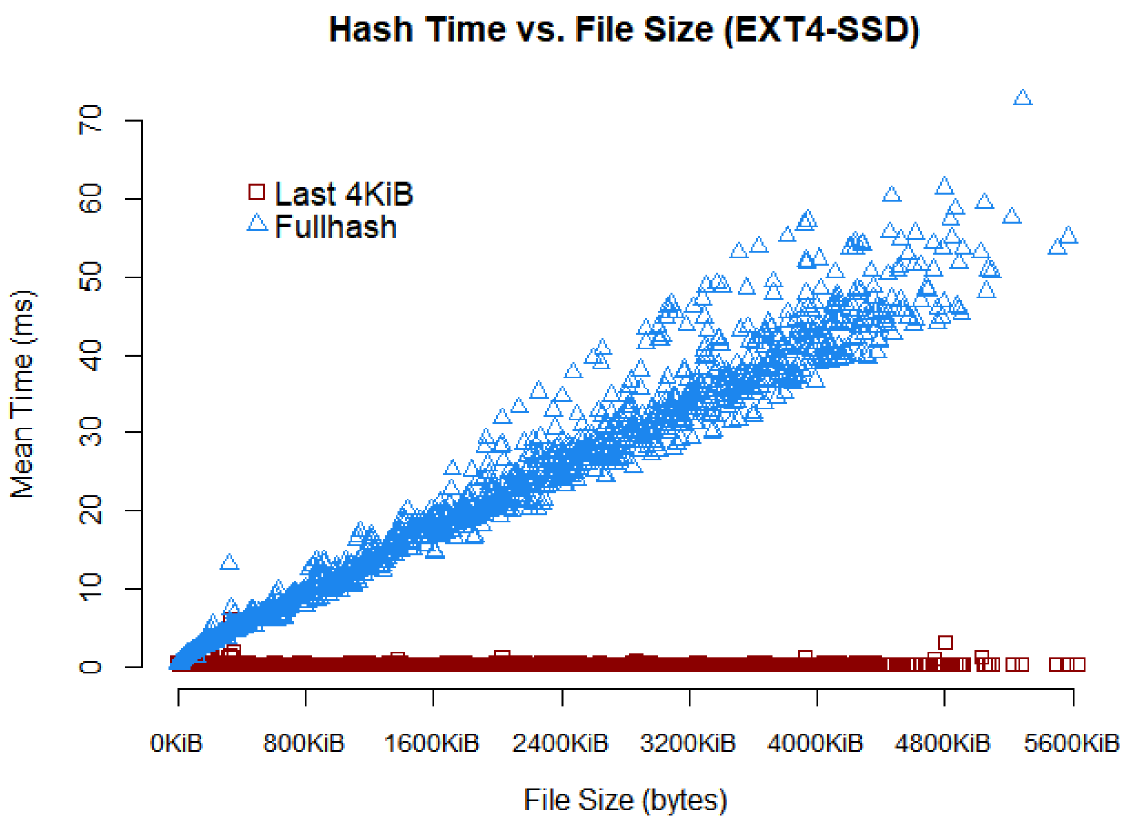


Fig. 4.2 File size plotted against hash time for Last 4KiB and full file hashing techniques. Mean file times are shown for file size bins of 8192 bytes, carried out single threaded on the SSD with EXT4.

File Size	Fullhash Time	Last 4KiB Improvement Factor
(Flickr) 123 KiB	2.5 ms	7.9×
400 KiB	5.4 ms	17.0×
800 KiB	9.6 ms	30.1×
1 MiB	12.0 ms	37.4×
(Gov. PNG) 1.4 MiB	16.2 ms	50.8×
10 MiB	108.4 ms	339.4×

Table 4.5 Linear Regression predictions for the time taken to read and hash full files of various sizes, with the relative benefit of Last 4KiB hashing. Figures are derived from the same data as Figure 4.2 using the SSD with EXT4. Improvement factor is calculated by dividing the Fullhash time by the constant time to acquire and hash Last 4KiB (0.32ms).

The time taken to acquire the Last 4KiB of each file is essentially a fixed cost which does not scale with file size, with a mean of 0.32ms in this experiment. Full file hashing, on the other hand, unsurprisingly scales linearly with the size of the file, easily taking tens of milliseconds to process when file sizes are in the mebibytes.

Linear regression was then used to generate a predictive model to obtain hash times and their respective Last 4KiB performance factors, depicted in Table 4.5. The equation for these predictions, where x is the byte size of the file, is as follows:

$$Fullhash\_time(s) = 1.02196 \times 10^{-8}x + 1.23139 \times 10^{-3} \tag{4.1}$$

These predictions match up fairly well with the SSD EXT4 benchmarks with 32 threads, and therefore serve as a reasonable indicator of performance. Files with 10MiB of data, which may include very high-resolution JPEGs, uncompressed images, or short videos, would enjoy a performance increase factor over 300×. This level of scaling means that when file sizes are in the Mebibytes, sub-file hashing strategies should perform much better than full file hashing, regardless of storage device or file system.

4.4.3 Confirmation Hashing

The Last 4KiB strategy generated unique signatures for all unique files across both datasets, however, some conditions may change this, with homogeneous data, similar files, or very large volumes of images. This can be tackled by reading more data from the end of the file, as with the Last 12KiB approach, or using the less performant First+Last 4KiB strategy. In some use cases it may still be prudent to fully hash the entire file to confirm the original detection assessment. In order to assess the impact of confirmation hashing, worst case scenario estimates are provided in Table 4.6 for 1%, 10% and 33% confirmation rates. These estimates ignore disk or memory caching for the file, which in reality may reduce the confirmation overhead.

Even at a 33% confirmation rate, where 33% of files are fully hashed, the only scenario which is slower than full file hashing is on the hard disk with Flickr 1 Million. The best case for Govdocs PNG and EXT4 is still 220% faster than fully hashing all files, with the worst SSD case for Flickr on NTFS still being 20% faster.

As with the sub-file Huffman technique, confirmation hashing need not be carried out at the initial detection stage, and can be conducted during idle processing times. It is also important to reiterate that, based on the discrimination results in Section 4.3, the evidence suggests that the Last 4KiB approach does not need this verification step.

Dataset/ Device	Threads	Last 4KiB Time (s)	Hit Rate (% Positive Hits)			Fullhash Time (s)
			1%	10%	33%	
Flickr 1 Million						
NTFS HDD	1	1605.6	1618.4 (0.8×)	1746.9 (0.7×)	2174.8 (0.6×)	1284.9
NTFS SSD	32	319.7	327.4 (2.3×)	404.0 (1.9×)	659.3 (1.2×)	766.7
EXT4 SSD	32	37.9	40.5 (6.4×)	66.4 (3.9×)	152.7 (1.7×)	259.1
Govdocs PNG						
NTFS HDD	1	665.9	683.7 (2.6×)	861.4 (2.1×)	1453.0 (1.2×)	1776.8
NTFS SSD	32	33.6	39.6 (15.4×)	100.5 (6.1×)	303.3 (2.0×)	609.0
EXT4 SSD	32	4.4	7.5 (41.4×)	38.6 (8.1×)	141.9 (2.2×)	310.5

Table 4.6 Fullhash confirmation impact on the Last 4KiB sub-file approach. 1%, 10% and 33% of corpus assumed to generate a positive hit, respectively. Calculated by adding the appropriate percentage of Fullhash time to the Sub-file approach, and therefore ignores the benefits of caching. Relative improvement factors in brackets.

4.4.4 Comparison with File Type Specific Sub-file Signatures

The Last 4KiB approach and the file-specific approaches in the previous chapter have similar behavioural characteristics. Both approaches are close to the performance of the least costly logical sub-file access (reading the first disk block of a logical file), meaning that they introduce little performance degradation, while producing useful file signatures. For eight threads on the workstation's EXT4 SSD, Last 4KiB reaches a performance factor of 4.6× for Flickr 1 Million, and 49.9× for Govdocs PNG, while the stream based approaches achieve 4.8× and 45.7×, respectively. At higher thread counts on Flickr, Last 4KiB pulls slightly ahead of the Huffman approach, with 6.8× vs. 5.7×. The slight advantage in performance for Last 4KiB is likely because it reads a maximum of two disk blocks, while the streaming approaches occasionally have to process very large file headers. In this sense, the fixed upper limit on the cost of reading the Last 4KiB means that it will perform more consistently, regardless of the dataset.

The Last 4KiB approach also has the benefit of being highly discriminative, as it focuses on reading very high entropy compressed data from the end of the file. However, if a given file type does not end with compressed data, this approach will be far less effective. file-specific approaches, on the other hand, should be able to extract a reasonably effective signature from any file type, potentially at the cost of reading a slightly larger proportion of the mean file. The stream based approach also has the benefit of skipping irrelevant data which does not contribute to the perceptual content of an image file, while the Last 4KiB approach blindly hashes whatever data happens to be in the logical block. However, all sub-file approaches appear to be more robust than traditional hashing approaches, for both full file hashing, and block-based hashing.

The primary argument for using file-specific approaches is that they can extract signatures from features which are somehow representative of the entire file, as with optimised Huffman tables. This semantic link between signature and file content is more intuitive, and theoretically robust, than simply identifying high entropy offsets. However, when the use case is simply to identify exact file duplicates, there appears to be no reason not to favour the Last 4KiB approach, particularly as it does not produce any false positives in these datasets.

4.4.5 Outcome

The main contribution of Sections 4.3 and 4.4 is the development of a generic sub-file hashing approach which produces unique signatures at the million image scale. These signatures essentially offer constant time complexity for both extraction and lookup times, in a manner which scales very favourably to large file sizes. Hashing the Last 4KiB of each file produced a 70$\times$ performance increase over full file hashing on SSDs, again showing that substantial reductions in forensic processing time are possible on non-mechanical media.

The analysis of local media is not the only use case for contraband detection techniques. For this reason, the block-based sub-file approach is evaluated for networked storage media in the next section, before discussing the wider applicability of this approach to digital forensics in Section 4.6, with concluding remarks for the chapter in Section 4.7.

4.5 Generic Sub-file Approaches on Networked Storage Media

The forensic use case assumed thus far has been the analysis of local computers, or hard drives, where the forensic analyst has physical access to the device. However, cloud computing is increasingly a problem for forensics investigations [15, 18], as more and more illegal activity migrates to off-site servers. Indeed, the volume problem in forensics, which is the motivation for the literature discussed in Section 2.2, is no less pronounced in investigations of cloud and networked corporate assets. One example of this is the 2012 Megaupload case, where 150 terabytes of data was seized by the FBI [133]. It is therefore important that new forensics techniques are flexible enough to address problems generated in cloud and networked environments, particularly as targeted cloud forensics is in its infancy [109].

The work in this section explores the merits of applying the block-based sub-file signature approach to networked storage, both on a Local Area Network (LAN), and across the Internet to a virtual machine hosted on a cloud platform. The logical, file level, nature of this approach lends itself well to the current cloud landscape, as Roussev et al. [109] note that logical acquisition is the norm for current cloud based investigations. Similarly, as network based investigations will be limited by network bandwidth [110], data reduction approaches should prove promising for reducing bottlenecks. The work in this Section has also previously been published at the IEEE Cyber Security 2018 Conference [4], separately from the work in Sections 4.3 and 4.4. Supplementary materials are found in Appendix E. What follows is a brief discussion of current remote acquisition techniques in forensics, before describing the experimental set-up and results.

4.5.1 Remote Acquisition

The most common approach to analysing an electronic device is to isolate the storage media by physically removing it, then connecting it to a write blocker to acquire data from it. However, it may not be practical to seize large volumes of equipment due to physical storage limitations, easily damaged equipment, conflicting business requirements, or the inability to obtain cloud based devices. In these cases the fallback is to remotely collect evidence over a network connection for later analysis.

Scanlon and Kechadi [134] describe the Remote Acquisition Forensics Tool (RAFT), which makes use of a modified Ubuntu Live CD to acquire evidence and send it to a forensic server over the Internet. This approach is intended to expedite the forensics process by reducing the time taken to seize and physically transport evidence, but is limited by the bandwidth of the available Internet connection. There is also the requirement that the analysed machine be booted via the live CD. A similar approach is taken in Koopmans and James [135], which uses a similar live CD to execute automated hash lookups and string searches for the purposes of forensic triage. Collated evidence is transferred to a forensic server using NFS, with similar bandwidth limitations.

A proactive approach to evidence gathering may be taken by the server host in order to facilitate fast forensic analysis. Homem et al. [136] describe the Live Evidence Information Aggregation (LEIA), which makes use of a hypervisor on client machines and a peer-to-peer network for distribution, with cloud based storage back-ends. Known hashes and compression are used to reduce the amount of data to be acquired and the system is designed for scalable data aggregation and analysis. All systems to be analysed must run the hypervisor and connect to the peer-to-peer network for distributed processing.

In some cases it may not be practical to power off core business resources, meaning that evidence has to be acquired on running servers without modification to the infrastructure. Sealey [130] discusses a remote forensic acquisition process on live servers using the Encase Enterprise Edition software from Guidance Software. A servlet application is copied to a device, which then extradites data across the network. This method has the added benefit of not requiring physical access to the machine or to other networked devices.

A cloud specific approach, aimed at acquiring cloud storage services, such as Dropbox and Google Drive, is discussed by Roussev et al. [137]. This method takes advantage of the Application Programming Interface (API) endpoints exposed by cloud providers for client applications and third party integration. API based approaches allow for a more complete snapshot of evidential data, and can access revision history and content which has not been synced to the local device.

A key factor with all network based acquisition methods is the bandwidth available for uploading evidence. Many existing approaches rely on running software directly on the machine which is being investigated, with data reduction being achieved by pre-processing data locally on the device. In contrast to these approaches, the work in this section examines the case where a common interface is available for data being served from a file server, but does not assume that the examiner has physical access or can execute live programs on the device. Additionally, cloud based storage is treated as a logical, network mapped drive, making no assumptions about the availability of an API.

4.5.2 Experimental Set-up

The speed at which forensic evidence can be acquired from a networked device is dependent on the available network throughput, be it the speed of the local network, or the bandwidth of an Internet connection. Reducing the amount of data which is sent over the network will reduce the impact of this bottleneck, potentially resulting in lower processing times. The generic, block-based, sub-file hashing strategies in this chapter can effect data reduction while only requiring that a standard network storage protocol is available on the device.

Two sub-file strategies were chosen for these experiments: *i)* **Last n**, with values of *n* corresponding to 4KiB, 8KiB, 12KiB and 16KiB to assess the impact of various data block sizes on the network bandwidth bottleneck. *ii)* **First+Last n**, with the value of 4KiB for *n*, is chosen to assess the trade-off between data block size and data location, which may have different characteristics across a network. The First *n* strategy is omitted as it has been shown to be less discriminative at scale.

Three datasets were used, corresponding to the 25,000 image subsets in Table 4.1. Subsets were chosen to allow for manageable experiment run-time and cloud storage costs, with each subset increasing the file sizes to assess the file scaling performance of networked acquisitions. For both sub-file techniques, if the file size is smaller than the requested block data the entire file is requested and hashed. Early experiments showed that additional overheads from checking file sizes for all files caused overheads which slowed the acquisition process. As file sizes are rarely smaller than the read blocks, it is faster to catch any read exceptions for these very small files, rather than request the size of all files.[29]

Two experimental scenarios were chosen to reflect typical methods of accessing files across a network. *i)* Via a LAN connection, with both a 1Gbps and 100Mbps connections being tested, and *ii)* a VPN connection over the Internet to a remote file server, with a 100Mbps symmetric connection for the client. The LAN server in the first scenario was on an isolated network and connected to the client using a high performance switch, while the Internet server was hosted on a London based Digital Ocean droplet and accessed via a university network. Both scenarios are depicted in Figure 4.3.

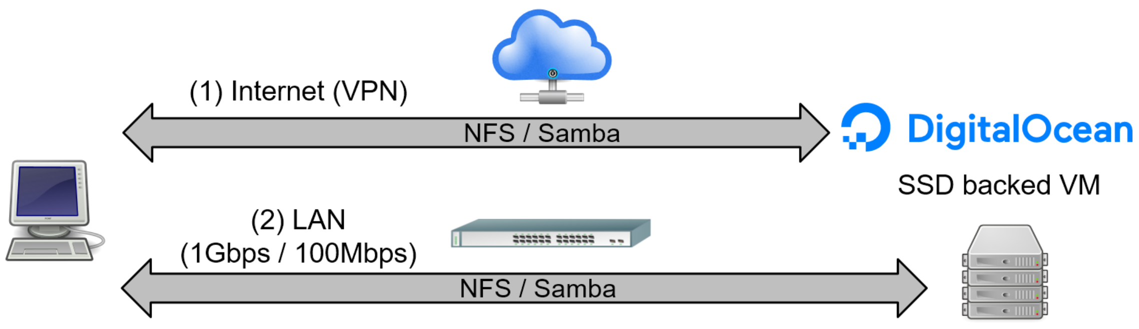


Fig. 4.3 Experimental set-up for block-based sub-file benchmarking of networked storage.

[29]There was little to no performance degradation from checking file sizes on local disks, however.

Timed benchmarks were performed for both scenarios using the Samba and NFS file serving technologies in order to assess their potential impact on the performance of sub-file hashing strategies. As neither Samba nor NFS provides encrypted transport, a VPN tunnel is required in order to securely access files across the Internet. As such, in order to maintain a realistic access scenario, OpenVPN [138] was used to create a secure connection. All software was left with default configurations from a fresh install. Table 4.7 provides hardware specifications and software versions for both scenarios. While not all enterprise storage solutions are expected to make use of SSDs, as with the servers in this work, they are expected to have high performance, enabling them to serve many users simultaneously.

Benchmark code was written in Python 2.7, using the built-in `hashlib` library to generate SHA256 hashes for data blocks. File read orders were determined using Python's `os.listdir` function, and volumes were mounted read only by the client. The client machine ran Windows 7 Enterprise, while servers made use of Ubuntu. This setup made use of NFS version three, while the servers were capable of running version four. Reported times do not include file enumeration from `os.listdir`. The mean value of three repeated runs was taken for each set of parameters, with client side memory caches being cleared before each run[30]. The same code was used for both local and network based generic sub-file benchmarks, with the exception of directory names for scripts, with snippets provided in Appendix E.1.

Preserving File Access Times

As the client device is not interacting with the target file system directly, instead interfacing with a layer of abstraction in the form of a file server, no guarantees can be made that file system metadata will remain unchanged. Despite the Windows client having read only access, the default configurations in this experiment caused file access times to be modified in the EXT4 directories containing the test data. Without access to the server configuration it cannot be verified that modification will not occur. It is therefore recommended that available file metadata be collected prior to remotely hashing files on a file server to prevent the loss of potentially useful forensic information.

4.5.3 Timed Benchmark Results

As there are many variables the analysis of benchmark results focuses on the various factors affecting performance. Generally speaking, trends hold across network connection types and file server software. An overall summary of the performance of the sub-file

[30]Using the same EmptyStandbyList tool [122] from the PNG experiments in Section 3.4

Machine	Specification	Software
Client Workstation	HP EliteDesk 800 G1 i5-4590s, 4GiB RAM	Windows 7 Enterprise 64bit NFS v3 client OpenVPN Client (AES 256/SHA256, no compression, 2048bit key)
LAN Server (Hypervisor)	2× Intel Xeon E5-2697v4 384 GiB RAM RAID 10 10× 840 EVO 1TB SSD	vSphere 6.5.0 ESXi 6.5.0
LAN Virtual Machine	ESXi 6.5+ Virtual Machine 1 Virtual CPU, 2GiB RAM 500GB SSD storage (EXT4)	Ubuntu 17.04 LTS Samba server 4.3.11-ubuntu NFS server 1.2.8-9.2ubuntu2
Internet Server	Digital Ocean $10 Droplet 1 Virtual CPU, 2GiB RAM 50GB SSD storage (EXT4)	Ubuntu 16.04 LTS Samba server 4.5.8-ubuntu NFS server 1:1.2.8-9ubuntu12.1

Table 4.7 Specifications of the equipment and software set-up for the networked generic sub-file benchmarks.

techniques relative to full file hashing is provided in Table 4.8, while a visual comparison between techniques is provided in Figure 4.6.

Thread Count and File Server

Multi-threaded file requests can maximise the throughput from an IO device, such that it is always kept busy with concurrent requests. The thread scaling performance of file serving protocols will be limited by the software implementations on both client and server, as well as the physical limitations of the network and underlying physical storage configuration.

Figure 4.4 shows the thread scaling performance of both NFS and Samba for the Govdocs PNG dataset on the 1Gbit LAN and Internet connections. This graph is representative of the behaviour for all tested connection set-ups and datasets, with Last 4KiB being chosen to represent all sub-file approaches for clarity.

The thread scaling of NFS is much more pronounced than Samba, typically performing relatively poorly for 1–2 threads, and surpassing Samba thereafter. This holds for both the sub-file and full file hashing approaches, which both begin to level off around 4 threads. Samba's implementation appears to be single threaded, which is a limiting factor when

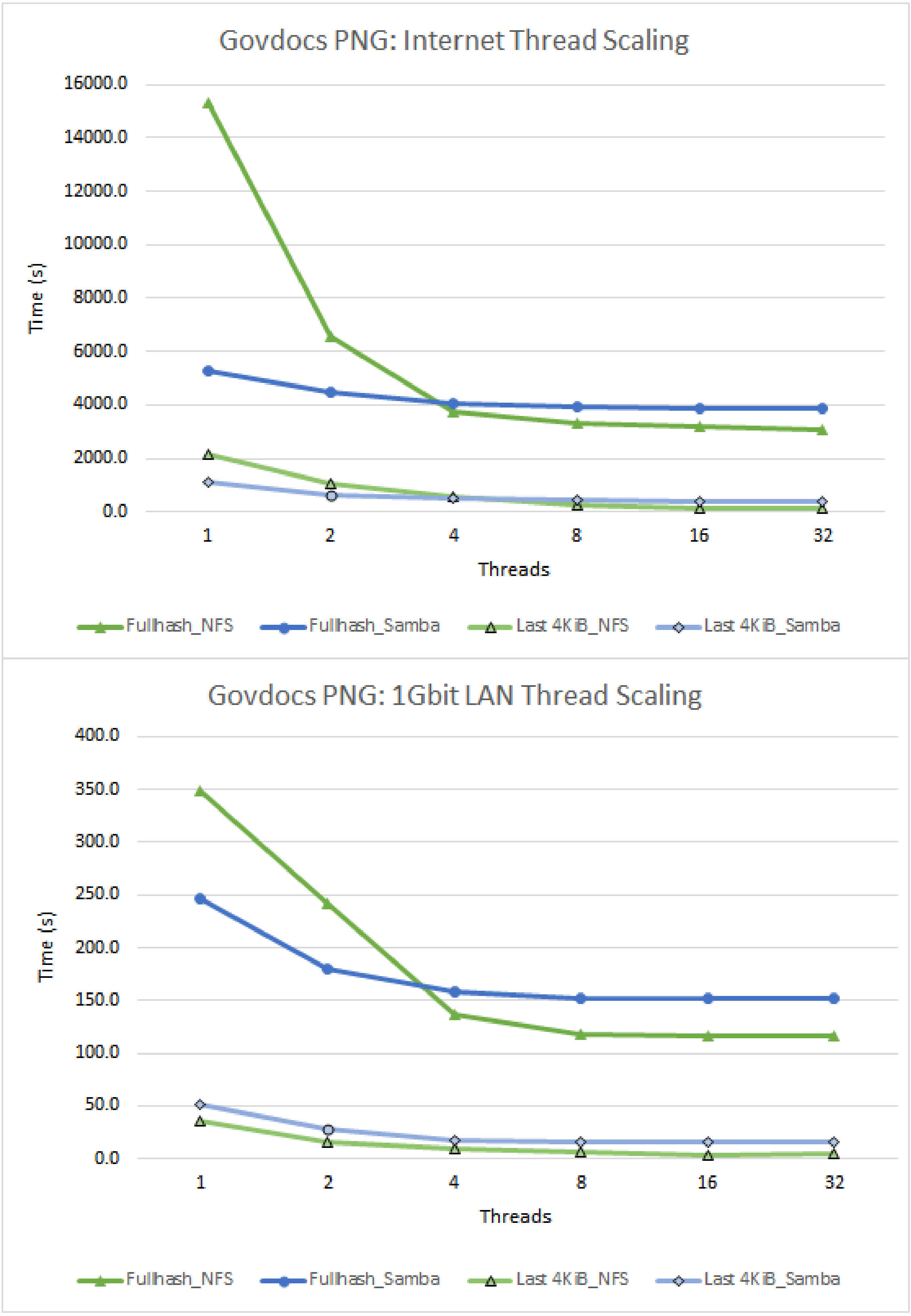


Fig. 4.4 The impact of thread count on the performance of the Last 4KiB and Fullhash techniques for the Govdocs PNG dataset on the 1Gbit LAN and 100Mbit Internet connection. Relative values are representative for all three datasets and LAN configurations.

increasing the number of concurrent requests. Despite this, both file servers see a benefit in all cases of up to 32 threads over the Internet, and 16 threads over the LAN connection, with performance degrading slightly at 32 threads over LAN. In all tested cases, the sub-file approach typically receives 2–3× the benefit of increasing from 1–32 threads than full file hashing.

These results show that a minimum of four threads should be used when hashing files from NFS and Samba servers, and that NFS outperforms Samba for both approaches at high thread counts. It should be noted that the overheads introduced by the network file system technologies are likely more complicated than those of local device file systems. In the networked case, the file server is a layer of abstraction on top of the file system used by the disk volume. The server also has to reply to external requests via its own protocol, which may be another source of overhead, with the implementation of both the client and server potentially introducing another potential bottleneck.

Performance Scaling With File Size

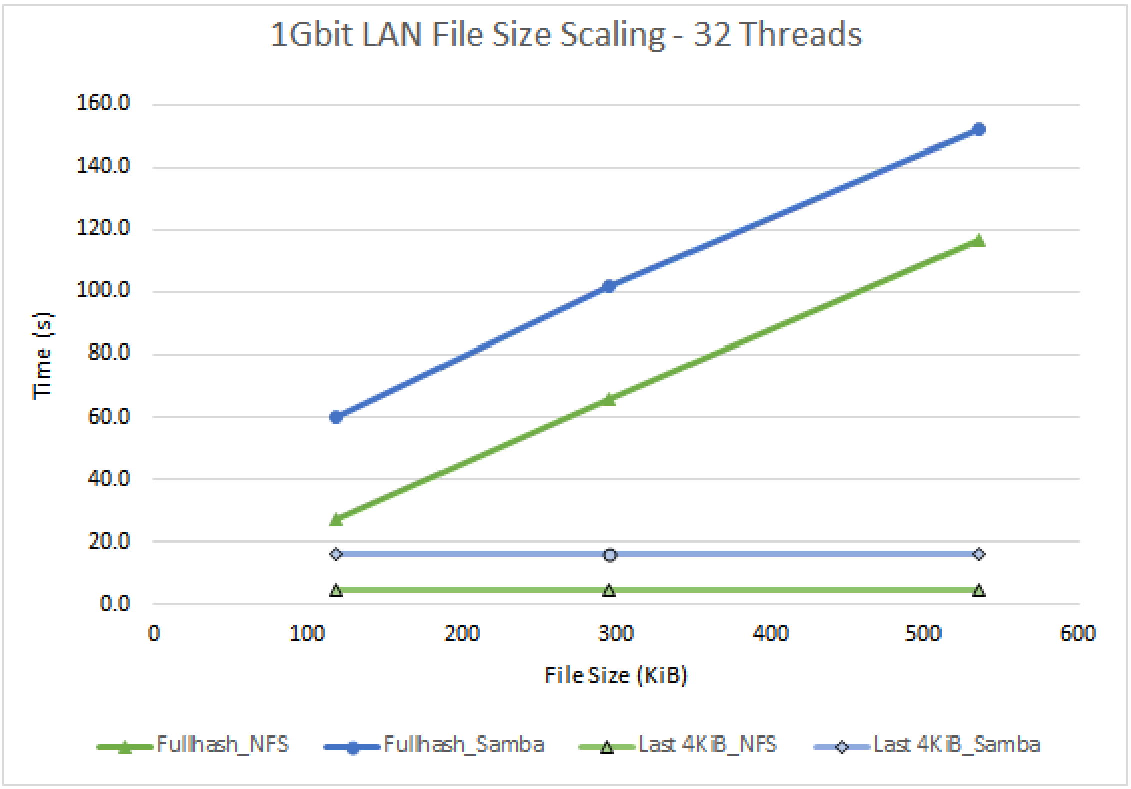


Fig. 4.5 The scaling of the 32 thread performance of Last 4KiB and Fullhash across each dataset for the 1Gbit LAN connection. Data points are for mean times and mean file sizes for each dataset. Relative values are representative for all network configurations.

One of the benefits of the sub-file signature approach in this work is that their processing costs should be independent of file size. That is, reading a 4KiB chunk of a file should take the same length of time regardless of the size of the file. Figure 4.5 shows this to be the case for both of the network file serving protocols. This observation also holds for the First+Last 4KiB approach.

Full file hashing appears to scale linearly with file size on networked file systems, which is in line with the observations made for local disks in Section 4.4.2. This linear scaling means that the gap in performance between full file hashing and sub-file approaches increases with file size, with the time difference already being substantial at the mean file size of 535KiB for the Govdocs PNG subset. For larger files, such as high-resolution photos, or even video files, sub-file approaches would again tend towards a tiny fraction of the processing time of full file hashing.

This file size scaling effect is amplified as the total throughput of the network is reduced. Going from 1Gbit to 100Mbps on the LAN connection increased full file hashing times by approximately 10×, while Last 4KiB only suffered a 2–3× performance penalty, with a 3–6× penalty for Last 16KiB. Full file hashing is typically bandwidth constrained at high thread counts, while the sub-file approaches are limited by the per-file response time of the entire request, with latencies accumulating all the way from the disk drive to file server response. However, increasing the block size used in the sub-file approaches also has a corresponding bandwidth penalty, which is discussed further in the following section.

Sub-file Hashing Techniques Compared

Figure 4.6 depicts the processing times for sub-file approaches across each connection type and file server. When bandwidth is not the primary bottleneck, the performance of all sub-file approaches converges at 32 threads, as in Figure 4.6a. When bandwidth is a limiting factor, as with Figure 4.6b, the volume of data to be transferred becomes a bottleneck, with each additional 4KiB block in the Last *n* technique resulting in a small performance penalty. The 8KiB of data hashed by First+Last 4KiB places its performance between Last 8KiB and Last 12KiB on Samba, while being very close to Last 8KiB performance on NFS.

However, as noted in Section 4.5.3, bandwidth is not the only limiting factor, with the total round trip of each transaction, i.e. from the client request to the client receiving the data for each file, having a particular impact on sub-file approaches. Figure 4.6c for the 100Mbit Internet connection shows behaviour more similar to the 1Gbit LAN (3a) scenario than to the equivalent 100Mbit bandwidth of the LAN connection (3b). A single transaction may involve multiple round trip requests at the file system level, as file handles

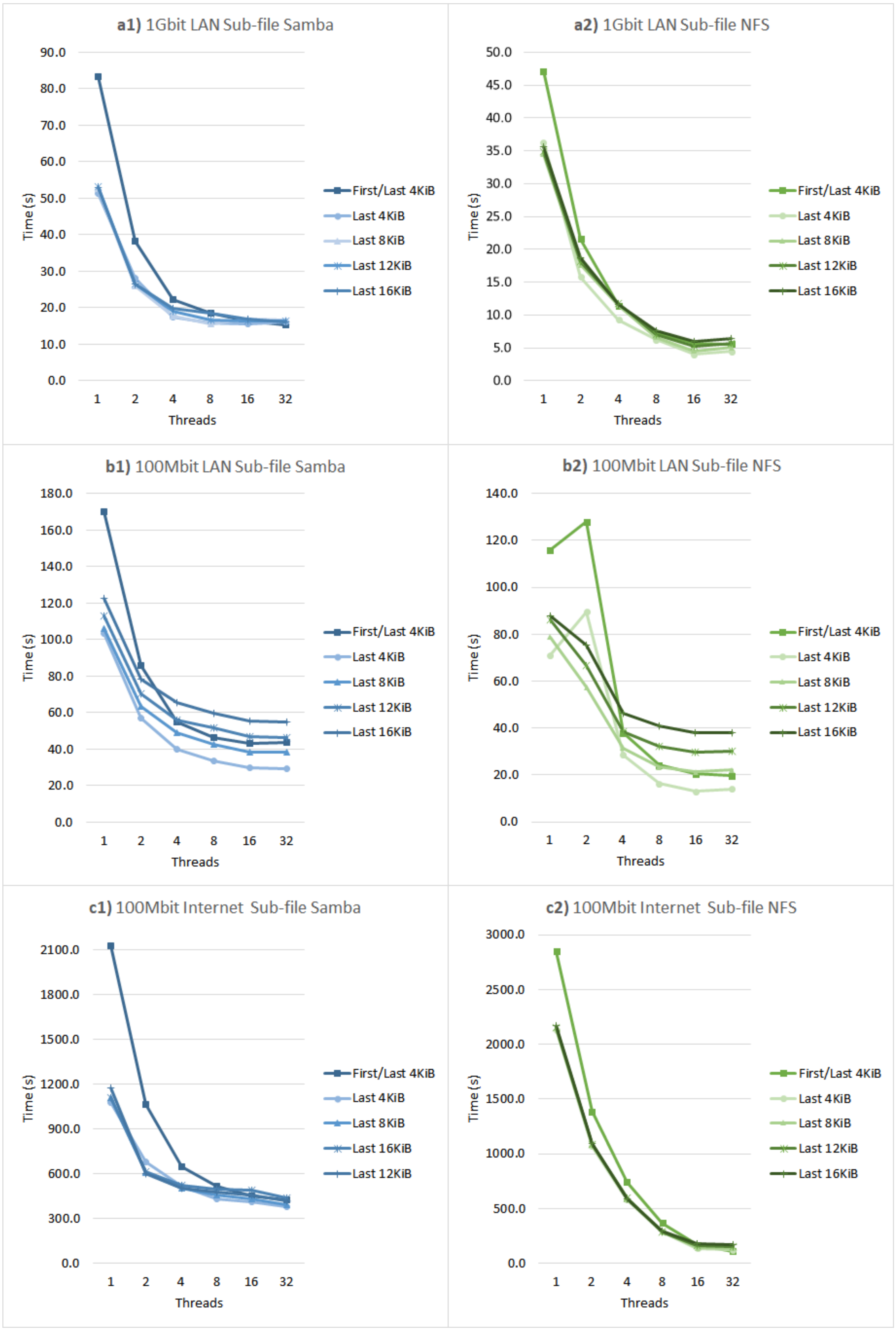


Fig. 4.6 Sub-file techniques compared across connection types. Samba on the left, NFS on the right. Mean times are the same across all datasets.

are opened, and file seeks are performed. As a result, the fraction of the transaction which is attributed to transferring the small data blocks is proportionally less. This difference is not attributable to the underlying storage media as both the LAN and Internet servers were theoretically able to saturate their network connections, with 4KiB random read performance from the SSD storage measured at 1.1Gbit/s and 175.7Mbit/s respectively, using the *iops* tool [139].

The cost of each transaction also explains why the Internet connection is an order of magnitude slower in acquiring data than the 100Mbit LAN scenario, despite having the same theoretical bandwidth. The fixed cost for a single request is increased due to the packets needing to be routed across the Internet, and thus the minimum file access time for any block size is increased. Given that the data block size is no longer an issue in this case, larger block sizes may be used, potentially increasing accuracy, at no effective cost in processing time. The opposite is true in heavily bandwidth constrained environments, where the smallest data block should be used.

Sub-file Hashing vs. Full File Hashing

Dataset	**Technique**	**Samba Performance Factor**			**NFS Performance Factor**		
		1Gbit LAN	100Mbit LAN	100Mbit Internet	1Gbit LAN	100Mbit LAN	100Mbit Internet
Flickr	Last 4KiB	3.72	10.20	3.13	6.18	18.29	6.77
	Last 16KiB	3.75	5.42	2.75	4.23	6.77	5.07
	First+Last 4KiB	3.95	6.81	2.79	4.94	13.06	7.55
Flickr PNG	Last 4KiB	6.31	23.45	6.13	14.84	45.54	14.75
	Last 16KiB	6.37	12.46	5.25	10.17	16.84	10.30
	First+Last 4KiB	6.70	15.64	5.49	11.87	32.51	16.39
Govdocs PNG	Last 4KiB	9.44	41.62	10.22	26.43	82.37	24.27
	Last 16KiB	9.52	22.10	8.99	18.10	30.46	18.37
	First+Last 4KiB	10.01	27.76	9.14	21.13	58.81	27.79

Table 4.8 A table of relative performance factors to Fullhash for each technique. Values presented are for 32 threads.

A performance summary of the tested sub-file techniques relative to full file hashing is provided in Table 4.8 for all 32 thread configurations.

All sub-file approaches prove viable in all scenarios, with the gigabit LAN and Internet connection performance being roughly equivalent for all sub-file methods, approximately 3–10$\times$ depending on the dataset for Samba, and 5–25$\times$ for NFS. However Last 4KiB

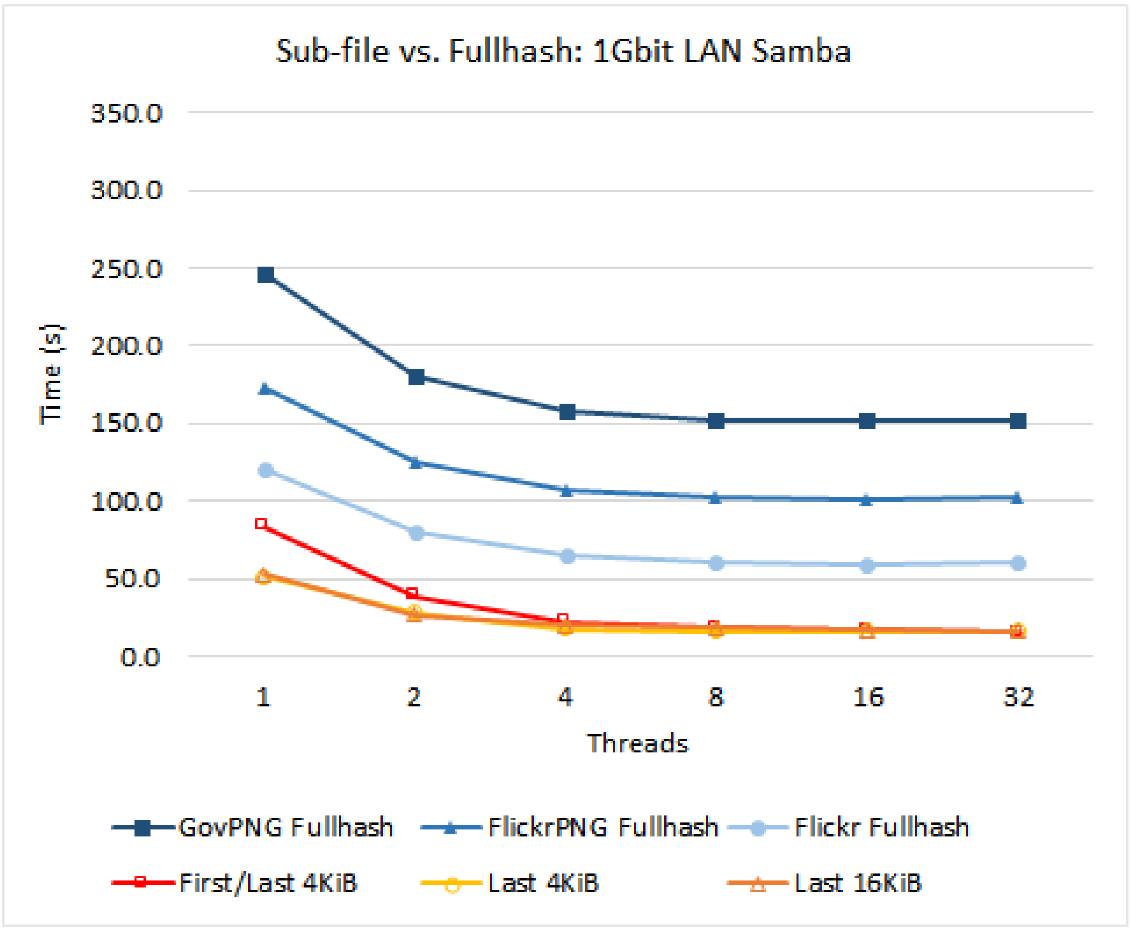


Fig. 4.7 Sub-file techniques compared with full file hashing each dataset. Only shown for 1Gbit LAN on Samba.

has the clear advantage when bandwidth is the limiting factor as with the 100Mbit LAN connection, reaching up to 41× on Samba, and 82× on NFS. Increasing the data block size to 16KiB results in roughly half the performance in this scenario, while First+Last 4KiB is typically around 2/3 of the speed.

Sub-file hashing strategies are capable of substantial reductions in the time taken to detect contraband on a networked file server, even on the relatively small file sizes used here. Figure 4.7 shows the relative cost of hashing each dataset, where file sizes roughly double in between datasets. An estimate based on this figure would suggest that the mean file would have to be tens of KiB for the sub-file approach to perform as well as full file hashing, when bandwidth is not a constraint. This is likely unrealistic in real-world use, and suggests that the sub-file technique is always beneficial when accessing files logically over a network connection. However, if this approach is to be used as part of an API based approach, it is likely that additional overheads will be introduced, which may affect the applicability of the technique for small file sizes. That being said, as storage performance continues to increase, and overheads are reduced, the burden will be increasingly placed on the overall network bandwidth, which is where the Last 4KiB technique is particularly effective.

4.5.4 Confirmation Hashing

1Gbit LAN NFS		**Performance Factor**		
Technique	**Hit Rate**	Flickr	Flickr PNG	Govdoccs PNG
Last 4KiB	0%	6.18	14.84	26.43
	1%	5.82	12.92	20.90
	10%	3.82	5.97	7.25
	33.3%	2.02	2.50	2.70
Last 16KiB	0%	4.23	10.17	18.10
	1%	4.01	9.23	15.33
	10%	2.97	5.04	6.44
	33.3%	1.76	2.32	2.58
First+Last 4KiB	0%	4.94	11.87	21.13
	1%	4.71	10.61	17.45
	10%	3.31	5.43	6.79
	33.3%	1.87	2.40	2.63

Table 4.9 The impact of confirming detected contraband with full file hashing at detection rates of 1%, 10% and 33.3%. Values are only shown for NFS on the 1Gbit LAN connection.

Sub-file approaches are designed for fast contraband detection when dealing with large data volumes, while providing a high degree of accuracy. While not strictly necessary for these sub-file approaches, a full file hashing verification step can be performed when contraband is detected. As such, the projected worst case scenarios for confirmation hashing at various rates are provided in Table 4.9 for completeness. Again, these suggested detection rates are likely much higher than the base rate of detection in real cases. The performance impact of confirmation hashing increases with file size, however even with a high detection rate of 33% sub-file processing is still 2–3$\times$ faster than full file hashing.

4.5.5 Outcome

This section demonstrates that sub-file hashing strategies can be used to rapidly investigate remote networked storage. Experiments were performed on Samba and NFS servers over the Internet and two LAN configurations, showing up to an 82× performance increase over full file hashing on standard resolution image files. Sub-file techniques were shown to perform better on NFS than Samba, with limited bandwidth over a 100Mbit LAN creating a dramatic gulf in performance between sub-file and full file hashing approaches.

The physical storage media is typically the performance bottleneck for forensic processing. However, in a world where large quantities of data are being stored remotely on cloud services, network performance may present itself as a frequent bottleneck in the forensic process. Sub-file hashing can be used to greatly decrease the time taken to perform an investigation over a network, which is particularly important when dealing with hundreds of terabytes of data on large scale storage networks.

4.6 Generic Sub-file Signatures in Practice

This chapter has shown that generic, block-based, sub-file signatures are both highly discriminative, to the point of being unique in large datasets, and substantially faster than full file hashing, particularly on SSDs and over network connections. The remainder of the chapter discusses the implications of this approach for real-world use cases.

4.6.1 Discriminating Power and Data Reduction

The Last 4KiB technique successfully discriminates between all unique files in two large datasets by exploiting high entropy compressed data streams, which typically run until the end of the file. This approach should work for any compressed file type, except those which potentially end with a large amount of low entropy data in a file footer. However, the author is not aware of any such file formats at this time, meaning that this approach has good generalisability. When comparing hashes generated by very similar files, some matches may be made on non-identical files if the data towards the end of the file is the same. This is analogous to the problem encountered when hashing PNG data blocks in Section 3.4.3. However, this depends on the nature of the changes, as some compression schemes may have knock-on effects from file modifications. One example of this is how optimised Huffman tables change depending on the content of the file. If the last few scan lines in the image remain the same, but the rest of the image changes, then the frequency of DCT codes would change, resulting in a different Huffman table and an altered compression stream. Such modification would cause the Last 4KiB approach, as well as any other technique based on binary identity or similarity, to fail. This technique essentially operates at the compression level, meaning that a re-compressed, or re-encoded, file with the exact same perceptual content would not be detected. That is, the hash is related to a small portion of the file content, but not directly to the perceptual content.

The Last 4KiB approach effects slightly better data reduction than the JPEG and PNG approaches in Chapter 3, as no more than 4096 bytes are read from disk, even if it is effectively spread across two disk blocks. This is equivalent to around 3% of the mean Flickr 1 Million JPEG, and 0.3% of the mean Govdocs PNGs. Of course, file access overheads are essentially unchanged from the discussion in Section 3.6.3, meaning that the effective performance benefits over full file hashing are capped by the storage technology. Increases in storage random 4KiB read performance will have a corresponding improvement in performance for both generic and file type specific sub-file approaches.

4.6.2 Network Based Detection

The block-based approach to sub-file hashing works well when accessing files via a network file system. The nature of the approach means that, as long as the means of accessing files across a network allows for fractional, logical, file access, this method should reduce file processing times significantly. However, the underlying assumption is that the protocol overheads are relatively low, and that they do not overwhelm the benefits of reading small parts of a file. The technique could also be applied to other file serving technologies, such as FTP and WebDAV, though the impact of overheads for those technologies would have to be empirically verified. A brief examination of the potential improvements on cloud storage platforms, exemplified by Dropbox, showed that it was possible to use the http `range` operator to get partial files when making a request to the Dropbox API. However, the overheads of accessing individuals files in this way mitigated the majority of the benefit. At 8 threads on the Govdocs PNG subset, Last 4KiB only effected at 30% improvement over full file hashing. However, there may be ways of reducing these overheads, and further work is required.

When monitoring a network stream, the generic sub-file approach is likely not going to be very useful. This is because the Last 4KiB technique needs to capture the last 2 blocks of file data by chance when sampling from the stream. When a file contains n data blocks, this means that there are n-2 blocks in the file which are not the Last 4KiB of the file, such that the larger the file, the lower the probability of capturing the appropriate file segment. In practice, packet sizes are typically much smaller than 4KiB, with a standard Maximum Transmission Unit (MTU) size of 1500 bytes (1.465KiB) meaning that each 4KiB of file data is approximately three packets on a network. The file marker-based approaches in Chapter 3 are therefore more appropriate for this use case. Alternatively, traditional block-based hashing, where all blocks in the file are hashed should also work, as it can monitor all parts of the file, even if some of the blocks are repeated across files [44]. This, again, assumes that there are no block alignment issues with packet sizes, or that file blocks can be reconstructed.

4.6.3 Digital Forensics Process

From the standpoint of a forensic investigation, the majority of the discussion pertaining to file-specific sub-file approaches (Section 3.6.5) also applies to generic sub-file techniques. Signature generation techniques in both chapters require logical file access, and would be most effective in the triage stage of an investigation, paired with the data reduction approach from Grier and Richard [43]. The main difference appears to be the higher potential discrimination from the Last *n* approach, particularly when the block size is increased. This would mean that there is likely no need to perform confirmation hashing when detecting contraband, though MD5 and SHA1 hashes would likely still be of use when reporting findings. Additionally, as the Last *n* approach should work on all, or most, media files, it is perhaps more practical for real-world investigations, while the JPEG approach requires a specific type of JPEG, and PNG files require verification.

File size considerations are still important on hard disk drives, as very small files receive little benefit. However, with careful handling of read orders, it is possible that the Last 4KiB approach may be used on all files on a device without any performance degradation. On solid-state media, it seems that Last 4KiB will be faster in every scenario, and would therefore be deployable on all evidential devices.

Finally, as with any new signature generation technique, new contraband databases would have to be constructed from source images. This would mean both having access to an existing corpus of illegal images to generate signatures for comparison, and also generating signatures for newly discovered files. However, this is no more of a problem than updating file hashes to use a newer cryptographic hash, or similarity hash, such as those discussed in Section 2.3.

4.6.4 Non-forensic Use Cases

All sub-file approaches have the potential to be used for detecting identical files in non-forensics contexts. One such use case is that of file de-duplication, particularly across a network for large scale corporate storage. In this case, it would be prudent to perform confirmation hashing to guarantee that the file is not a small modification of the original.[31] This is an effective way of quickly identifying duplicate files while requiring minimal bandwidth. Of course, this approach is unnecessary if block-level de-duplication technologies are used when storing files [140].

Similarly, sub-file signatures may be useful for detecting copyrighted media, however, they are less useful in this space as the focus is likely shifted towards identifying perceptual, semantic, content, rather than binary level features.

[31] In a forensic context, a small variant of the original is still relevant to the investigation, but it is an important distinction for file de-duplication.

4.7 Conclusions

This chapter has successfully developed generic sub-file hashing strategies for contraband detection, which can significantly reduce processing times on local and remote storage media. Signatures generated from the end of each file were shown to be unique for two large datasets, while only requiring 1–3% of a small file to be read. These sub-file techniques perform similarly to the file-specific approaches in Chapter 3, while extending to a much wider range of file types.

In a world where high-resolution digital cameras and Internet connections are incredibly pervasive, digital investigators are struggling to keep up. The sub-file approaches proposed in this chapter present a means for an investigator to significantly reduce the amount of data which is required to be processed, and boost processing speeds by orders of magnitude on large files. Even on relatively small files, there was a measured benefit of up to a 70$\times$ processing time reduction on an SSD, and up to 80$\times$ over a network connection. Sub-file approaches are accurate, and process all relevant files, allowing for a high degree of confidence and speed. These strategies were evaluated favourably against the criteria in Section 3.2, and can help alleviate investigative backlogs which are crippling law enforcement agencies across the UK.

Chapter 5

Thumbnail-based Contraband Detection

5.1 Introduction

Solutions to the volume problem in forensics are essentially split into two categories: *i)* Parallel processing at disk read speed, and *ii)* Data reduction approaches. The sub-file approaches in Chapters 3 and 4, focus on file level data reduction, and in doing so also define an alternative method of processing files at "disk speed". In contrast, most data reduction approaches operate at the disk level, rather than the file level, such that many of the files on a given storage media are never processed. Disk level data reduction is achieved in the literature by randomly sampling from the disk [29], or by deriving samples from file system metadata using heuristics for selection [43]. An alternative approach in this chapter effects rapid forensic triage by exploiting centralised thumbnail stores on the Windows operating system. This allows contraband to be detected while only requiring a tiny fraction of the disk to be read, processing a single point of the file system for each user on a device.

Windows operating systems generate image previews by downscaling an image to create a smaller version, a thumbnail, which is then stored in a cache for later use. These thumbnails are typically displayed when utilising a file system browser, such as Windows Explorer. Since Windows Vista, these caches have been stored in a centralised location for each user, with sizes on the order of hundreds of mebibytes, regardless of the capacity of the disk. Thumbnails are essentially a reduced representation of the full-sized image, and can be used as a proxy when searching for contraband. The thumbnail cache itself acts as a reduced representation of images on a computer, allowing for very fast initial disk triage to be performed at a low cost, in a matter of seconds.

The primary focus of this work is the Windows 10 thumbnail cache. Existing work in this area, discussed below in Section 5.2, focuses on cataloguing the behaviour of thumbnails on previous Windows versions, but does not explicitly use them for triage purposes. This chapter updates the existing information on thumbnail caches to include the Windows 10 operating system and provides a comparison of differences between thumbnails on different Windows versions. Two methods of exploiting these caches are then discussed, before examining how thumbnail analysis may be useful on cloud storage services.

5.2 Thumbnail Forensics

Before examining how thumbnail caches can be used for rapid contraband detection, a brief overview of the existing literature is provided below. The first section discusses thumbnail caches on Windows, followed by forensic work done on centralised caches in other operating systems.

5.2.1 Centralised Thumbnail Caches in Windows

Morris and Chivers [31] note that a paradigm shift for storing thumbnails occurred between Windows XP and Windows Vista, with a centralised thumbcache structure for each user replacing the thumbs.db files in each directory. This means that instead of thumbnails being found in the same directory as their source files, they are now located in a per-user store, located at:
`[Drive]:/Users/[Username]/AppData/Local/Microsoft/Windows/Explorer/`

Separate database files store thumbnails of various sizes, named thumbcache_*xxx*.db, where *xxx* specifies the maximum thumbnail dimension for each side. For Windows Vista and Windows 7, these are 32×32, 96×96, 256×256, and 1024×1024, though thumbnails need not be square. Each cache entry contains some metadata and the thumbnail image itself, which is usually stored in either the JPEG, BMP, or PNG format. Cache entry IDs are mapped back to their source file via the thumbcache_idx.db file. Images viewed with the Windows Explorer preview pane generate 1024 pixel thumbnails unless the images are small enough to fit a smaller thumbnail size. 256 pixel thumbnails correspond to extra-large icons, with smaller sizes being used for smaller previews. Thumbnail generation is not limited to images, with documents such as PDFs and file system directories also having thumbnail preview options. Directory previews look like an open folder with multiple pages, each corresponding to an item in the directory. Viewing a directory icon may trigger thumbnail creation for files within the directory without the user previewing them directly. Additionally, entries are created for files on removable media in centralised thumbnail cache. Thumbnails may be removed from the system using the Disk Clean-up utility built-in to Windows, and thumbnail generation can be disabled using group policies.

Windows 8 expands on the thumbnail dimensions included in previous versions, adding 16×16, 48×48, and WIDE thumbnails to support additional start menu functionality [141]. Additionally, iconcache_*x*.db files were added, with corresponding dimensions matching all thumbcache files.

Parsonage [142] explores the behaviour of thumbnail generation on Windows Vista and Windows 7 further, finding that legacy thumbs.db files are still being generated when accessing files using the Universal Naming Convention (UNC), which specifies a hostname, share name, and optional file path. Additionally, they note that thumbnails may be created in a variety of circumstances which do not involve the original source image being opened or viewed in thumbnail mode. This includes thumbnails being automatically generated based on the two most recently modified files in a directory, for use by the directory preview. Thumbnails are also generated as a result of dragging and copying files, even when such activities are cancelled by the user.

Prior work has not described the changes made to the thumbcache in Windows 10, which is presented here in Section 5.3. Additionally, this chapter discusses the differences in thumbnails generated across Windows versions in Section 5.5.2.

5.2.2 Centralised Thumbnail Caches in Other Operating Systems

Centralised thumbnail caches are not used exclusively in Windows operating systems. Morris and Chivers [143] discuss the centralised thumbnail caches used in versions 9.10 and 10.04 of the Kubuntu and Ubuntu operating systems, respectively. Thumbnails are stored in the user's home directory in `~/.thumbnails`[32], which is composed of three sub-directories: fail, large, and normal. All thumbnails are of the PNG format, and correspond to 128×128 pixels for normal thumbnails, and 256×256 for large. Thumbnails which are unable to be generated are tracked using the 'fail' sub-directory. In contrast to Windows, these thumbnail caches have no index, instead, thumbnails are simply stored in these sub-directories, with MD5 hashes of the source URI for filenames, allowing for fast thumbnail lookups. These thumbnails contain the modified times of the original files, such that updated originals signal that the thumbnail must be rebuilt. Importantly, as the thumbnail cache in Ubuntu is just a directory, it may be used by third-party programs, thus there is a possibility that some thumbnails may be stored with different parameters from those generated by the operating system.

Thumbnail generation, storage, and extraction for the Android operating system are explored by Leom et al. [144]. The thumbnail cache for the built-in Gallery application is

[32]This appears to have moved to ~/.cache/thumbnails in more recent releases of Ubuntu.

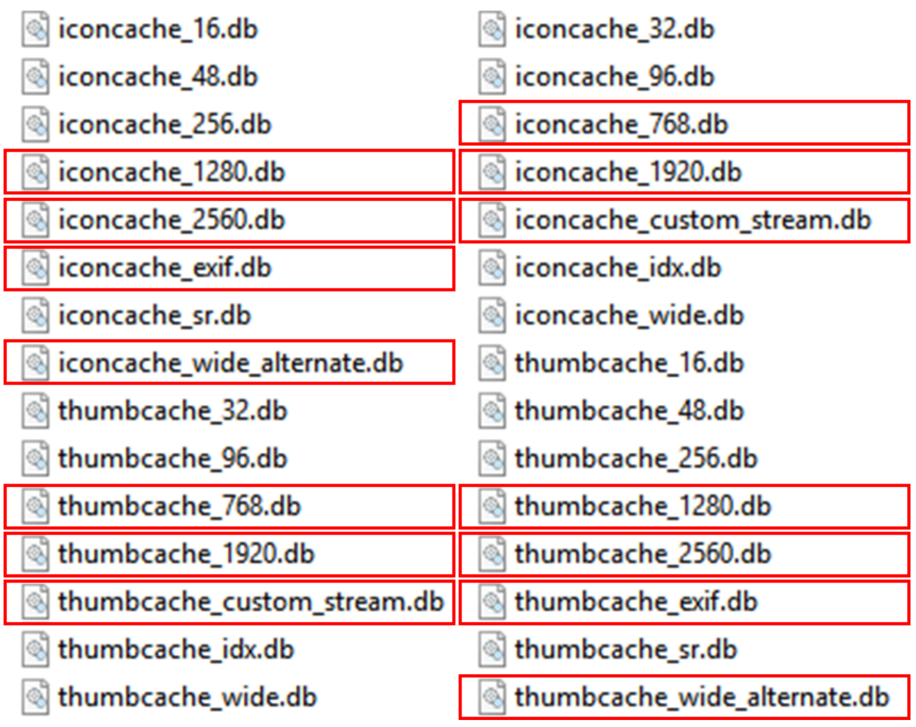
located at:
`/sdcard/Android/data/com.google.android.gallery3d.cache/imgcache.0`
and is composed of a single file which contains thumbnails of 200×200 and 640×480 pixels in the JPEG format. Larger thumbnails are generated for most pictures when they are created using the camera application, while smaller thumbnails are created when viewing images in the Gallery application.

Finally, Newcomer and Martin [145] describe the per-user thumbnail caches found on Mac OS X. Thumbnails for the Spotlight and Finder applications are provided by the QuickLook technology and are generated for many file types. Different thumbnail sizes are used depending on the finder viewing mode. The directory containing the cache is located in `/private/var/folders/<2random>/<30random>/C/com.apple.QuickLook.thumbnailcache` for versions 10.7 and 10.8, with the 'C' in the path being switched for '–Caches–' on the older 10.5 and 10.6 OS. Each user has their own directory in folders consisting of two random lowercase characters, potentially including an underscore, followed by a subdirectory of 30 random characters. It is unclear if this is actually stochastic or deterministically generated by a hashing function. An SQLite database, index.sqlite is used to store information on the location of each source file, as well as the location of its thumbnail, which is indexed by referencing its offset in the corresponding thumbnail.data binary file containing thumbnail image data.

5.3 The Windows 10 Thumbnail Cache

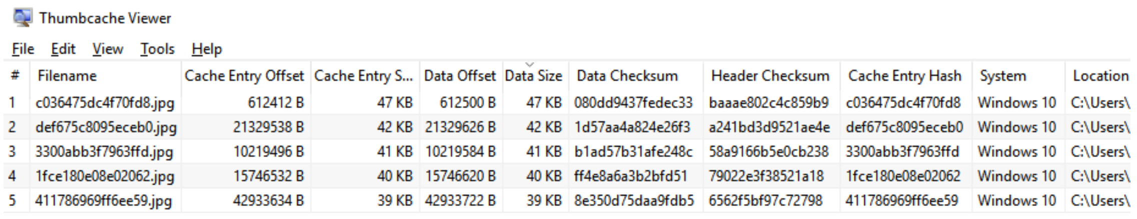
While the Windows 10 thumbnail cache remains in the same location as its predecessors, several changes have been made to the files used to store the thumbnails. A complete list of thumbcache files for a Windows 10 user is depicted in Figure 5.1. The 1024×1024 thumbnail database was removed, replaced by 768×768 and 1280×1280 pixel thumbnails, with very large 1920×1920 and 2560×2560 thumbnails also being added. Wide thumbnails also have an extra file named thumbcache_wide_alternate.db, while completely new items are included in the thumbcache_exif.db, thumbcache_custom_stream.db, and respective iconcache files. The thumbcache_exif.db database appears to contain thumbnails which are embedded in source JPEGs, converted to use the quantisation tables of the other thumbnail databases. Example thumbcache entries, as displayed by the Thumbcache Viewer application [146] are provided in Figure 5.2.

This work focuses on the 96×96 and 256×256 thumbnails caches, as they are the most commonly generated, corresponding to small/medium/large and extra-large thumbnail previews. Thumbnails with dimensions of 96 are stored as bitmaps, while those with 256 pixel dimensions are stored as JPEGs. Version 1709 of Windows 10 (October 2017)



iconcache_16.db	iconcache_32.db
iconcache_48.db	iconcache_96.db
iconcache_256.db	iconcache_768.db
iconcache_1280.db	iconcache_1920.db
iconcache_2560.db	iconcache_custom_stream.db
iconcache_exif.db	iconcache_idx.db
iconcache_sr.db	iconcache_wide.db
iconcache_wide_alternate.db	thumbcache_16.db
thumbcache_32.db	thumbcache_48.db
thumbcache_96.db	thumbcache_256.db
thumbcache_768.db	thumbcache_1280.db
thumbcache_1920.db	thumbcache_2560.db
thumbcache_custom_stream.db	thumbcache_exif.db
thumbcache_idx.db	thumbcache_sr.db
thumbcache_wide.db	thumbcache_wide_alternate.db

Fig. 5.1 A screenshot of thumbcache files present on Windows 10. Files added in Windows 10 are indicated with boxes. iconcaches and thumbcaches of dimensions 1024×1024 were removed for Windows 10.



Thumbcache Viewer

File Edit View Tools Help

#	Filename	Cache Entry Offset	Cache Entry S...	Data Offset	Data Size	Data Checksum	Header Checksum	Cache Entry Hash	System	Location
1	c036475dc4f70fd8.jpg	612412 B	47 KB	612500 B	47 KB	080dd9437fedec33	baaae802c4c859b9	c036475dc4f70fd8	Windows 10	C:\Users\
2	def675c8095eceb0.jpg	21329538 B	42 KB	21329626 B	42 KB	1d57aa4a824e26f3	a241bd3d9521ae4e	def675c8095eceb0	Windows 10	C:\Users\
3	3300abb3f7963ffd.jpg	10219496 B	41 KB	10219584 B	41 KB	b1ad57b31afe248c	58a9166b5e0cb238	3300abb3f7963ffd	Windows 10	C:\Users\
4	1fce180e08e02062.jpg	15746532 B	40 KB	15746620 B	40 KB	ff4e8a6a3b2bfd51	79022e3f38521a18	1fce180e08e02062	Windows 10	C:\Users\
5	411786969ff6ee59.jpg	42933634 B	39 KB	42933722 B	39 KB	8e350d75daa9fdb5	6562f5bf97c72798	411786969ff6ee59	Windows 10	C:\Users\

Fig. 5.2 A screenshot of the Thumbcache Viewer application on a Windows 10 thumbcache. Data Checksum is a CRC64 of the entire thumbnail file, while Cache Entry Hash is the ID returned by the GetThumbnail method, and also serves internally as the file name. The truncated Location is the file path to the thumbcache file.

enforced a limit of less than 500MiB on each of these cache files[33], this limitation appears to have been lifted on later versions (tested on 1803, April 2018). In place of this file size limitation the cache is regularly cleared by the SilentCleanup task in Windows [147], which runs the built-in Disk Cleanup utility, cleanmgr.exe, with the `/autoclean %systemdrive%` parameters, resetting the cache and moving old database files to a ThumbCacheToDelete directory. The behaviour of SilentClean can be modified such that the thumbcache does not reset when it runs by making a change to the Windows registry. Further work is required to test if old thumbnails which have not been accessed for extended periods of time are removed, but the current assumption is that they are not.

Even where both the 96-pixel and 256-pixel thumbcaches are at the size limits imposed by Windows 10 version 1709, this corresponds to less than 1GiB of data to process per-user. That is, for a single user with a 1TB drive, the worst case for this approach requires that only 0.1% of the disk is sequentially read. Assuming a sequential read speed of 100MiB/s[34],, this would take approximately 10 seconds to acquire, as opposed to the 3 hours required to read the entire terabyte disk. This value will rise slightly if additional thumbnail sizes are processed, or the device has multiple user accounts.

5.4 Dataset Creation

The primary focus of this work is Windows 10, however thumbnails were also examined for the Windows Vista and Windows 7 operating systems to facilitate comparison between versions. The Flickr 1 Million dataset [113], composed of 1 million JPEG images, was used to explore thumbcache forensics at scale. However, no existing tool allowed for Windows thumbnails to be generated automatically, nor were there any existing datasets of Windows thumbnails available. In order to address this, a method was developed to extract Windows thumbnails for each Flickr image on each platform, as described below in Section 5.4.1. While this approach allowed for the generation of 96 and 256 pixel thumbnails on Windows 7 and Windows 10, only 256 pixel thumbnails were automatically generated on Windows Vista.[35] To compensate for this, a small number of 96 pixel thumbnails were examined manually for Vista. No modifications in the binary or pixel domain were made to the images.

[33]Maximum values of 350MiB for 96-pixel thumbnails and 460MiB for 256-pixel thumbnails were observed, which were verified by users on the community forum at https://answers.microsoft.com/en-us/windows/forum/windows_10-files/after-fall-update-windows-10-puts-a-maximum-size/6ad0a1e7-38c0-4547-9b8b-f7f3906c3a12).

[34]For reference, a modern consumer HDD can sustain sequential read speeds of approximately 150-200MiB/s, while consumer SATA3 SSDs peak around 500MiB/s.

[35]This appears to be caused by the Windows Vista API not properly supporting the thumbnail dimensions argument.

5.4.1 Thumbnail Dataset Generation for Large Datasets

In order to acquire data to analyse at scale, the Windows shell API was utilised to generate thumbnails for images by accessing the `IThumbnailCache` interface and calling the `GetThumbnail` method.[36] This can be used to force the generation of a new thumbnail, or obtain one which has already been cached for an image. A parameter allows for the size of the thumbnail to be controlled, though only one size at a time may be requested. The API then returns a memory mapped bitmap, regardless of the format the thumbnail is stored in (BMP or JPEG). As a result, without knowing which function Windows uses to save the memory mapped image to a particular file type, it is non-trivial to recreate the exact binary data stored in the thumbcache from the object the API returns. That is, thumbnails saved to disk in this fashion will not be identical with those stored in the cache itself, and cannot be used directly for cryptographic hash based contraband detection.

To work around this issue, thumbnails can be obtained directly by parsing each thumbache_*xxx*.db file after calling `GetThumbnail`. This is achieved by keeping track of the IDs returned by each call to `GetThumbnail` and extracting items in the cache with matching IDs. These IDs appear as ‘Cache Entry Hash’ in Figure 5.2, and are derived from a hash function using the volume GUID, NTFS FILEID, file extension, and last modified time [148]. For efficiency, the thumbcache_*xxx*.db should not be parsed after each thumbnail is generated. Instead, several hundred, or thousand, thumbnails should be generated at a time, with all of them being recovered from the thumbcache_*xxx*.db in a single pass. However, a small enough batch size should be chosen such that thumbnails are not overwritten before they are recovered, with a batch size of 5000 proving effective in this work.

The end result of this processing is a dataset composed of the exact binaries stored by Windows in the thumbcache database files. This dataset can then be further processed to generate contraband lookup databases, as described in Section 5.5. The code used to automatically extract thumbnails from a directory is available on Github [149]. A pertinent partial code snippet is included in Appendix F.1.

5.5 Detecting Contraband Thumbnails Using Cryptographic Hashing

With a dataset of thumbnails in the format used by the Windows thumbcache, a lookup database may be created by calculating cryptographic hash digests, such as SHA256, for each thumbnail. When examining a Windows computer, each thumbcache_*xxx*.db file

[36]https://msdn.microsoft.com/en-us/library/windows/desktop/bb774628.aspx

may then be parsed, and each entry in the cache hashed and checked against this database. This could either be done in memory while parsing thumbcache files, or after extracting thumbnail images to a directory.

Section 5.5.1 discusses the possibility of augmenting this approach using CRC64 checksums which are already present in the thumbcache files, while Sections 5.5.2 and 5.5.3 discuss pragmatic concerns for this approach.

5.5.1 Intermediate Lookups Using Embedded Checksums

The Windows thumbnail cache stores CRC64 checksums for each thumbnail, depicted as ‘Data Checksum’ in Figure 5.2. This presents the opportunity to use these embedded checksums in a lookup database in the same manner as cryptographic hashes, except that they need only be parsed as strings from the thumbcache, rather than calculated from binary data at runtime.

To create a CRC64 lookup database, the Windows thumbnail checksum can be generated by calculating a CRC64 for the first 1024 bytes, a second CRC64 checksum for all remaining bytes and then XORing both values to produce the final checksum. This algorithm was identified from the source code of the Thumbcache Viewer [146] and verified manually by comparing the output and embedded checksums in the thumbcache.

However, these checksums are more likely to have hash collisions than cryptographic hashing algorithms [150]. While no collisions were observed for Windows 10’s 256 pixel thumbnails, they did occur for the 96 pixel bitmaps. Three pairs and a triplet of non-identical 96 pixel thumbnails possessed the same CRC64 checksum while producing different SHA256 digests.

As the CRC64 algorithm produces clashes at the million image scale, CRC64 matches should be verified using a cryptographic algorithm, such as SHA256. In this sense, the CRC64 checksum can be utilised in the same manner as the PNG signatures in Section 3.4, acting as an initial filter for files. This would be faster than computing SHA256 hashes for all thumbnails in the thumbnail cache, but would require a hash and checksum pair to be stored for each thumbnail in the lookup database.

5.5.2 Thumbnail Differences Between Windows Operating Systems

Thumbnails from the same version of Windows were shown to be consistent across two computers for both Windows 7 and Windows 10, which also held when using virtual machines. This was verified by extracting thumbnails for all images in the Flickr 1 Million dataset for multiple computers and verifying that SHA256 image hashes were identical between machines. As such, hardware differences should not cause thumbnails to be generated differently.

However, an examination of the thumbnails produced by Windows Vista, Windows 7, and Windows 10, showed that binary identity cannot be relied upon across different Windows versions, even when produced on the same computer. Differences can be explained in terms of the Windows API used to generate the thumbnails by the operating system, which can undergo change over time, such as when a new operating system is released. Based on prior Windows API version numbering, major changes to the API occur when a new operating system is released, and as such, thumbnail cache behaviour is likely to remain stable within a given version of Windows.

The following discussion examines thumbnail differences between Windows versions in some detail, with findings that could potentially have implications for everyday investigations.

96 Pixel Thumbnail Differences

All 96 pixel thumbnails are stored in the BMP format in the cache, however, they are not generated in an identical fashion between Windows versions. Following the standard 64 byte BMP file header, Windows Vista and 7 use the 40 byte `BITMAPINFOHEADER` for the Bitmap Information Header, while Windows 10 uses the longer 124 byte `BITMAPV5HEADER`. Additionally, while Windows Vista and Windows 7 use no compression, Windows 10 makes use of bitfields compression, meaning that the raw binary data will not be directly comparable to prior Windows releases. Pixel data for thumbnails, after extraction from the BMP format, was shown to be frequently identical across Windows versions, however this was not the case for the entire dataset. Figure 5.3 depicts an instance where two Windows versions produce different bitmap data for the same input image, with differences being highlighted by the resemble.js library [126].

A further complication is the aspect of the image used to produce the thumbnail in the first place. Windows 7 was observed to derive the 96 pixel thumbnails from embedded EXIF thumbnails when available, rather than from the full-sized image. This can produce a thumbnail preview with a different aspect than the full-sized image, as the EXIF thumbnail is not always updated when an image is cropped or otherwise edited [151]. An example of this phenomenon is provided in Figure 5.4, where elements of the uncropped image are present in the EXIF and Windows 7 thumbnails, but are no longer present in the full-sized version. This behaviour was present for thumbnails generated by both the API calls and manual inspection of 96 pixel thumbnails on Windows 7, but was not reproduced in Windows Vista or Windows 10. Curiously, the 256 pixel thumbnails in Windows 7

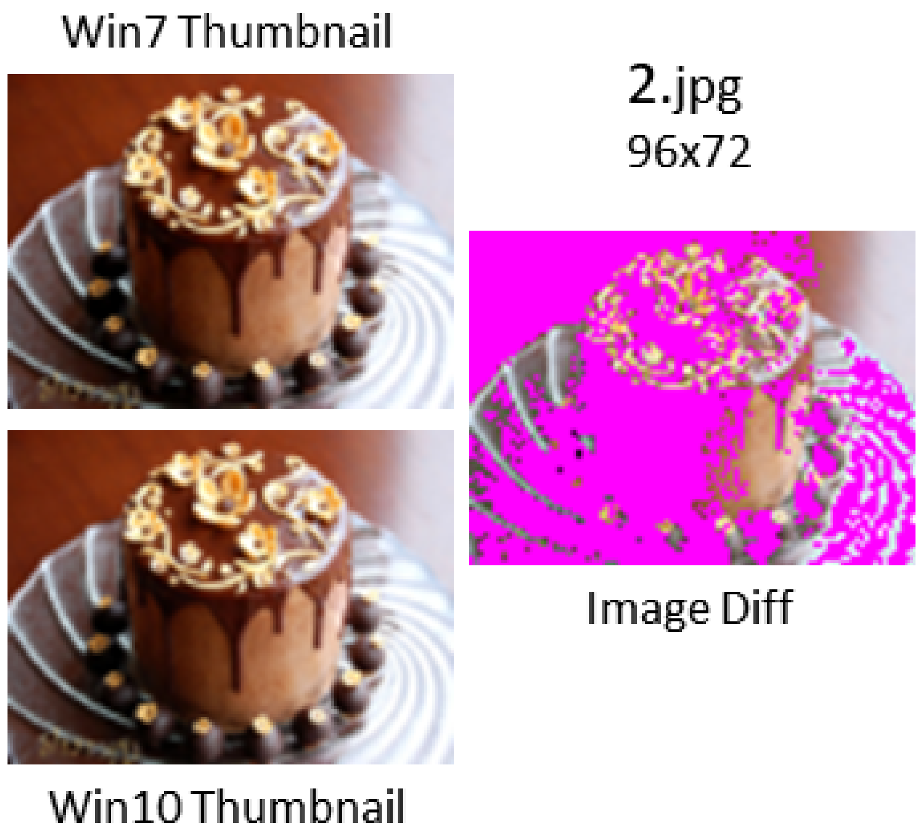


Fig. 5.3 An image diff of the Windows 7 and 10 thumbnails (96 pixel) generated for 2.jpg in the Flickr 1 Million dataset. Diff produced using the resemble.js library. Pixel differences in pink highlight that each Windows version may create slightly different thumbnail outputs.

are not generated from the embedded EXIF thumbnail, meaning that a user would see different image previews when switching between thumbnail sizes in Windows Explorer.

256 Pixel Thumbnail Differences

256 pixel thumbnails are stored in the JPEG format, which allows for varied compression parameters. Images across all three operating systems were found to use the default Huffman tables provided in the JPEG specification, however there are differences in the quantisation tables used. Windows Vista and Windows 7 share the same quantisation tables, while Windows 10 uses a table with finer quantisation on the higher DCT frequencies, resulting in higher image quality. This means that binary identity is lost as the thumbnails effectively have different quality settings. However, despite using the same quantisation tables, Windows Vista and Windows 7 also produce different binaries, with no images in the Flickr 1 Million dataset producing the same SHA256 digest across versions due to pixel differences. The reason for this difference cannot be attributed to the header and

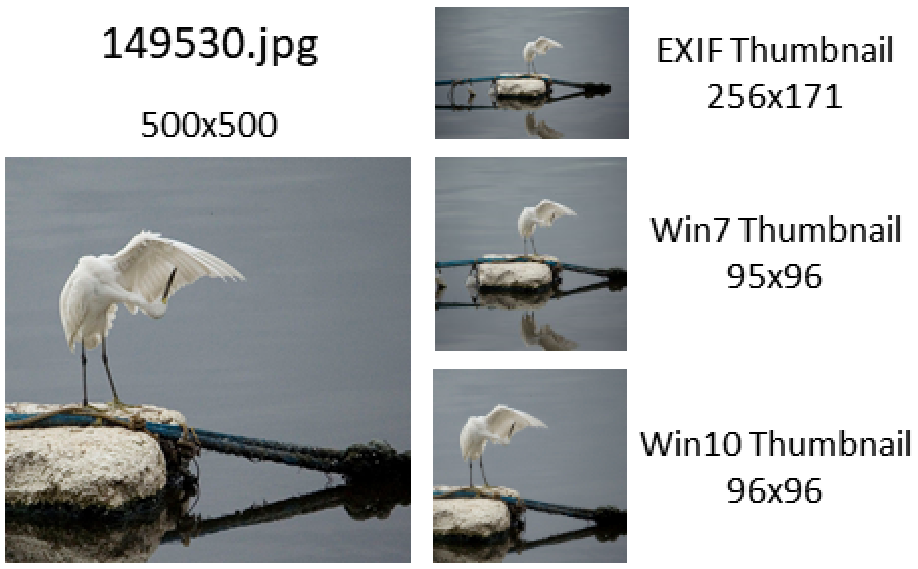


Fig. 5.4 A comparison of 49530.jpg from the Flickr dataset with its embedded EXIF thumbnail, Windows 7 thumbnail, and Windows 10 thumbnail. Image dimensions provided. Windows 7 uses embedded EXIF thumbnails to generate the 96 pixel thumbcache entries, which are not necessarily updated when an image is cropped or otherwise modified.

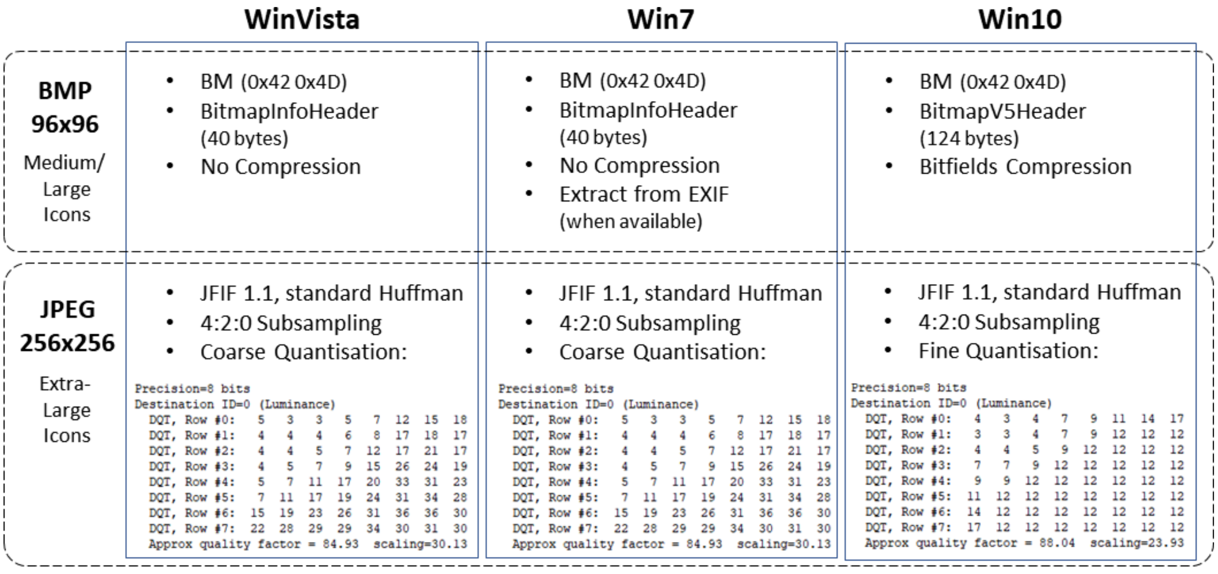
compression settings, which are identical on Windows 7 and Vista. The difference, then, must be attributed to the thumbnail API, introduced either in the rescaling of the image, or in the encoding of the JPEG.

A comparative overview of both thumbnail types is provided for the three tested Windows versions in Figure 5.5.

5.5.3 Dealing With Differences: Version Specific Databases

As the thumbnails produced by different versions of Windows frequently contain different binary data, it is not possible to create a universal lookup database from thumbnails generated on a single Windows release. For traditional cryptographic hash based lookups, one or more databases need to be created by generating thumbnails for each Windows variant. This would mean either a single unified lookup database, or a set of OS specific thumbnail databases, which may also contain CRC64 checksums. Additionally, thumbnails of each dimension must be hashed separately, increasing the total number of databases and fingerprints.

While this may add to overall maintenance overheads, its use for triage means that less commonly encountered operating systems need not be accommodated. Instead, the database could be maintained for the most popular, or most recent, Windows releases, which would make up the bulk of a typical investigator's workload.



	WinVista	Win7	Win10
BMP 96x96 Medium/ Large Icons	• BM (0x42 0x4D) • BitmapInfoHeader (40 bytes) • No Compression	• BM (0x42 0x4D) • BitmapInfoHeader (40 bytes) • No Compression • Extract from EXIF (when available)	• BM (0x42 0x4D) • BitmapV5Header (124 bytes) • Bitfields Compression
JPEG 256x256 Extra-Large Icons	• JFIF 1.1, standard Huffman • 4:2:0 Subsampling • Coarse Quantisation: Precision=8 bits Destination ID=0 (Luminance) DQT, Row #0: 5 3 3 5 7 12 15 18 DQT, Row #1: 4 4 4 6 8 17 18 17 DQT, Row #2: 4 4 5 7 12 17 21 17 DQT, Row #3: 4 5 7 9 15 26 24 19 DQT, Row #4: 5 7 11 17 20 33 31 23 DQT, Row #5: 7 11 17 19 24 31 34 28 DQT, Row #6: 15 19 23 26 31 36 36 30 DQT, Row #7: 22 28 29 29 34 30 31 30 Approx quality factor = 84.93 scaling=30.13	• JFIF 1.1, standard Huffman • 4:2:0 Subsampling • Coarse Quantisation: Precision=8 bits Destination ID=0 (Luminance) DQT, Row #0: 5 3 3 5 7 12 15 18 DQT, Row #1: 4 4 4 6 8 17 18 17 DQT, Row #2: 4 4 5 7 12 17 21 17 DQT, Row #3: 4 5 7 9 15 26 24 19 DQT, Row #4: 5 7 11 17 20 33 31 23 DQT, Row #5: 7 11 17 19 24 31 34 28 DQT, Row #6: 15 19 23 26 31 36 36 30 DQT, Row #7: 22 28 29 29 34 30 31 30 Approx quality factor = 84.93 scaling=30.13	• JFIF 1.1, standard Huffman • 4:2:0 Subsampling • Fine Quantisation: Precision=8 bits Destination ID=0 (Luminance) DQT, Row #0: 4 3 4 7 9 11 14 17 DQT, Row #1: 3 3 4 7 9 12 12 12 DQT, Row #2: 4 4 5 9 12 12 12 12 DQT, Row #3: 7 7 9 12 12 12 12 12 DQT, Row #4: 9 9 12 12 12 12 12 12 DQT, Row #5: 11 12 12 12 12 12 12 12 DQT, Row #6: 14 12 12 12 12 12 12 12 DQT, Row #7: 17 12 12 12 12 12 12 12 Approx quality factor = 88.04 scaling=23.93

Fig. 5.5 A comparison of thumbnail images for Windows Vista, Windows 7, and Windows 10. Chrominance quantisation tables are omitted for brevity.

5.5.4 Timed Benchmarks

To be effective, thumbnail-based triage must be very fast. As such, timed benchmarks were executed to assess the potential of this approach.

The first 10 images (0.jpg to 9.jpg) from the Flickr 1 Million collection were chosen to serve as known lookup items, with the corresponding 96 and 256 pixel thumbnails being used to generate SHA256 and CRC64 fingerprints. To create a realistically sized lookup database, a further 4,999,990 randomly generated SHA256 and CRC64 strings were included, creating four lookup databases of 5 million fingerprints each. This value was chosen to be in line with prior work discussing the real size of law enforcement databases [45]. Databases were loaded into memory for constant time lookups. In practice, this constant lookup time meant that there was no measurable difference between the lookup times for databases containing 10 items and 5 million items.

Two thumbnail cache files were then populated using the `GetThumbnail` method (as described in Section 5.4.1), with the first 10,000 images in the Flickr 1 Million dataset (integer file name order) being cached for thumbcache_96.db and the first 25,000 for thumbcache_256.db. These quantities were chosen as they are near the maximum observed capacities of these files, while avoiding thumbnails being overwritten. This resulted in cache files of 257MiB and 362MiB, respectively, which were then copied to avoid further manipulation.

Two machines were selected to perform the comparison: *i)* The same workstation from the previous local disk experiments (Core-i5 4690k, 16GiB DDR3, 525GB Crucial MX300 SSD, 4TB Western Digital Red HDD), and *ii)* A low specification netbook (Atom

N450, 1GiB DDR2, 160GB Western Digital HDD (OS)). This allowed representative benchmark times to be acquired from both a relatively high specification machine, and a slower, legacy system. Disk benchmarks are provided in Appendix B.

Benchmarks were performed on the workstation using both the HDD and SSD, while the netbook used only its internal hard drive. Two lookup modes were tested: *i)* initial CRC64 lookups, with positive hits being verified with SHA256 (CRC+), and *ii)* SHA256 only (SHA), as described above in Section 5.5. A single thread was used to parse the thumbcache_*xxx*.db files and perform lookups in the database. The overall execution time does not include the time taken to read lookup databases into memory, as it is assumed that they would be contained in the forensic application's executable in practice. Each run was repeated 30 times, with the memory cache being cleared each time using the `EmptyStandbyList` utility [122].

The median benchmark times[37], shown in Table 5.1, indicate that this approach is very fast, regardless of the storage media used, taking approximately three seconds in the worst case on the workstation, and 11 seconds on the netbook. Initial CRC lookups offered no benefit when using the Workstation's hard drive, but reduced times by approximately 25% on the SSD, and 30–50% on the netbook. This is likely because the workstation is bottlenecked by the storage read speed, rather than CPU or memory limitations, while the netbook likely has some performance overhead when calculating hashes. As parsing the SHA-only approach is sufficiently fast in all cases, it is likely not worth the extra memory overhead, or database upkeep, to perform CRC lookups.

While there is no guarantee that contraband on a device will have a thumbnail in one of the corresponding thumbcache_*xxx*.db files, this technique is fast enough to be used as an inexpensive initial check in a forensics investigation. Additionally, this extraction and processing time does not change with the size, or number, of disks present in a device, as the time taken is related only to the number of entries present in the individual thumbcache_*xxx*.db files, and the number of users on each device. This approach can then be relied upon to be executed quickly, with the potential to immediately identify contraband. Even if no contraband is detected, very little time has been used, providing the opportunity to deploy additional triage techniques.

The code for benchmarking this approach is also available on the Github repository [149].

[37]Median rather than mean, to account for small variance due to background processes in Windows, particularly on the netbook.

	96px Thumbnails		256px Thumbnails	
	CRC+	**SHA**	**CRC+**	**SHA**
Workstation HDD	2.22s	2.20s	3.03s	3.02s
Workstation SSD	0.60s	0.81s	0.85s	1.28s
Netbook HDD	4.80s	6.66s	5.07s	11.16s

Table 5.1 Benchmarks for the parsing and lookup times for thumbcache_96.db and thumbcache_256.db, containing 10k and 25k images, respectively. CRC+ verifies initial CRC64 hits with SHA256 lookups. Reported values are the median of 30 runs.

5.6 Robust Thumbnail Lookups Using Perceptual Hashes

Cryptographic hashes are simple to calculate and have constant time database lookups. However, their rigidity in only matching exact binary content proves troublesome for the thumbnail variations found across Windows operating system versions. One method for de-coupling thumbnail lookup databases from specific Windows versions is to focus on detecting thumbnails which are visually identical, rather than checking for identity in the binary domain. This perceptual approach falls under the semantic approximate matching category in the literature (see Section 2.3.2). Perceptual hashing [40] techniques aim to generate robust signatures from visual features, providing tolerance to content-preserving changes in the binary data. This would allow a full-sized image to be compared directly to its corresponding thumbnail, regardless of thumbnail dimensions, compression differences, or source Windows version.

Perceptual hashing approaches typically perform well even when images have been rescaled. This means that images in the thumbcache_*xxx*.db files may be compared to a database of perceptual hashes generated from the original, full-sized, contraband images. No intermediate thumbnails, or databases, need to be generated, which frees this approach from the operating system API. Indeed, assuming a robust perceptual hashing technique, this method should be completely operating system indifferent, performing equally well on Windows and non-Windows platforms.

5.6.1 Choices of Perceptual Hash: Phash and Blockhash

While there are many approaches to perceptual hashing, with their own properties and weaknesses [52], many of the published approaches do not provide implementations. As such, a pragmatic approach was taken where two popular perceptual hashing techniques with open source implementations were chosen for this work. A brief description of each perceptual hashing method is provided below.

Phash: The Phash library [152] is an open source perceptual hashing library for performing image comparisons. However, the original codebase has not been updated in some time. As a result, a similar, more recent, implementation was chosen in the Python ImageHash library [153]. This library contains several perceptual hashing approaches, including ahash (average colour hashing), dhash (gradient tracking), whash (discrete wavelet transform), and a modification of the original Phash (discrete cosine transform). Based on initial testing, the modified Phash algorithm was chosen as it had the best performance on the thumbnail datasets. Frequency transformations such as the DCT used in Phash are able to capture essential properties of an image, and have proven to be effective in the literature.

Blockhash: The Blockhash algorithm [154] breaks an image into blocks and compares the mean colour values between blocks to create a signature. As the original paper does not provide an implementation, a third-party derivative implementation was used for this research [155].

Default hash sizes for both algorithms were used (64bit for Phash, 256bit for Blockhash). The distance between two hashes was calculated using the Hamming distance, which is simply the sum of bit differences between signatures. The normalised Hamming distance was then calculated by dividing this sum by the length of the hash digest. This produces a distance between 0 and 1, where 0 indicates an identical perceptual hash, and 1 indicates that all bits are different. Reported distances below refer to this normalised Hamming distance.

5.6.2 Determining Distance Thresholds For Matching Images

Ideally, visually identical images should produce the same perceptual hash digest, and hashes for almost identical images should only differ by a small number of bits. As the thumbnails for a single source image can differ across Windows versions, having a matching algorithm which tolerates small variations is necessary. One way to achieve this is to set distance thresholds for what constitutes an image match, while making sure that this threshold is small enough to avoid visually dissimilar images from being considered a match. For example, setting the threshold to $t = 0.3$ would mean that images with a perceptual hash bit difference of less than 3 in 10 would be considered a match, while anything above this is not a match.

In order to determine an appropriate distance threshold for image matches it was necessary to explore typical distances between unrelated images. Thresholds were evaluated for the full-sized images in the Flickr 1 Million dataset by calculating pairwise distances from each image to 50 random images in the dataset, with no repeated pairings. Duplicate binary files as determined by the SHA256 algorithm were not included, resulting in a sample of slightly less than 50 million pairwise comparisons, of the potential 500 billion comparisons.

For both perceptual hashing algorithms, the mean and median normalised Hamming distances were almost exactly 0.5, and appear to be normally distributed. Plots for these distributions are provided in Figure 5.6 for Phash, and Figure 5.7 for Blockhash. As images in the Flickr dataset should be unrelated[38], these comparisons should hold for any heterogeneous dataset. This result likely reflects design choices behind the algorithms, where two random images should produce hashes which are around 50% different on average.

Based on this data, it is possible to derive false positive rates for various distance thresholds for unrelated Flickr 1 Million images. In this context, a false positive occurs when two non-identical images in the dataset register as matches for a given distance threshold. If this occurs often, then the distance threshold is too high, however, setting the threshold too low may exclude some legitimate matches, generating false negatives.

False positive rates in this dataset are provided for various thresholds in Table 5.2. As there are only expected to be tens of thousands of potential thumbnails on a device, given the SilentCleanup task [147], a relatively high false positive rate may be acceptable. A false positive rate of 0.01% would generate approximately 5 false positives for every 50,000 thumbnails, which places a little burden on a human examiner when positive hits are manually verified. Distance thresholds were calculated from actual data, rather than statistically from fitting normal distributions to the curves. This means that despite having a nearly identical mean and standard deviation, the distance thresholds for each hashing method are quite different. This can be explained by the lower utilisation of the normalised Hamming space by Phash, partially due to it using a lower number of bits per hash (64bit vs 256bit). However, this is a feature of using the default hash size for Phash, and, as such, is not corrected for statistically.

False Positive Rate	0.00001%	0.0001%	0.001%	0.01%
Phash Distance	0.1250	0.1875	0.2188	0.2500
Blockhash Distance	0.0469	0.0781	0.1172	0.1641

Table 5.2 False positive match rates and their corresponding Phash and Blockhash distance thresholds for pair-wise comparisons in the full-sized Flickr 1 Million dataset.

[38]Though there is at least one case where an image has identical pixel data but a different SHA256 hash digest.

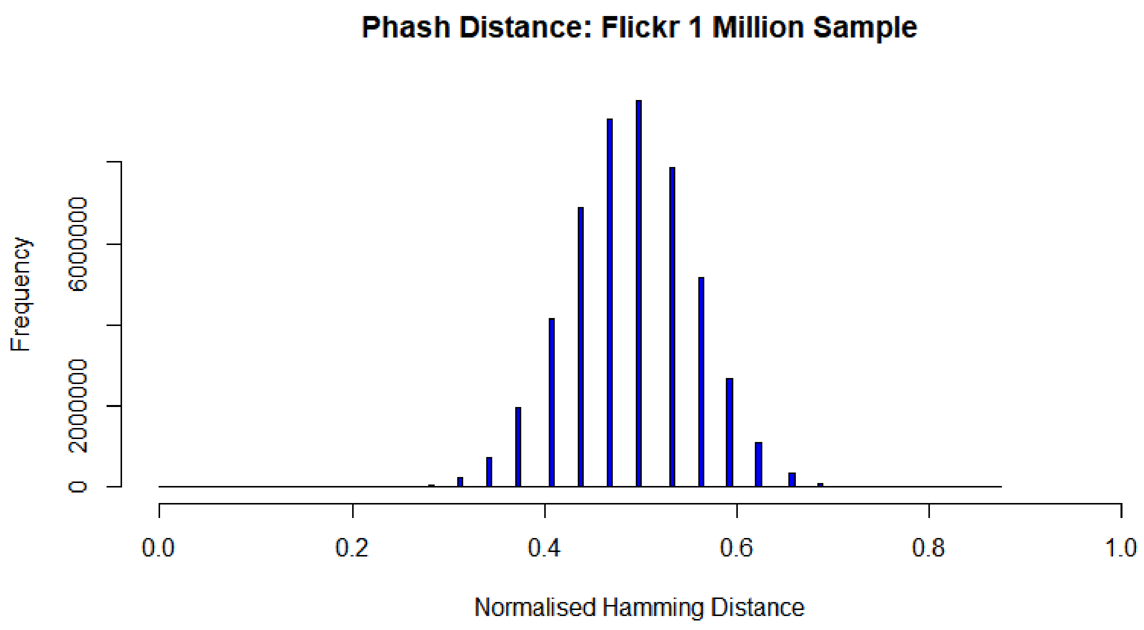


Fig. 5.6 Phash normalised Hamming distance distribution for the 50 million sample comparisons of the original Flickr 1 Million dataset. Distances values appear to have fewer discrete values than those for Blockhash, resulting in small gaps between bars.

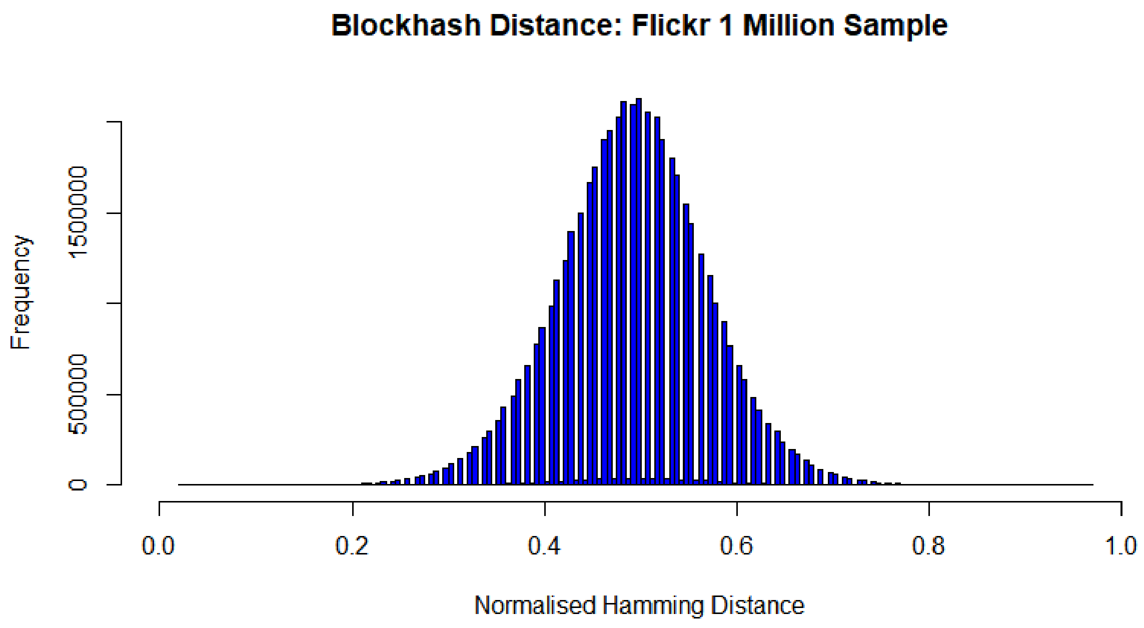


Fig. 5.7 Blockhash normalised Hamming distance distribution for the 50 million sample comparisons of the original Flickr 1 Million dataset.

False Negative Rate		
	Phash	**Blockhash**
Distance	0.2500	0.1641
Win10 96px	0.0494	0.0467
Win10 256px	0.0227	0.0404

Table 5.3 False negative rates for each perceptual hash algorithm when comparing full-sized Flickr 1 Million images to their Windows 10 thumbnails. Distance thresholds are set for a 0.01% false positive rate.

Using a false positive rate of 0.01%, the effective false negative rates for detecting image thumbnails were calculated. A false negative occurs when a full-sized image and its thumbnail is not considered to be a match, and is caused by the distance threshold being set too low. In the context of an investigation, this would mean that an item of contraband was not automatically detected, even though it is present in the thumbnail cache. False negative rates for Windows 10 are provided in Table 5.3, with values for Windows Vista and Windows 7 being almost identical. Both algorithms were found to miss 1 in 2000–4000 thumbnails. This can be attributed to both weaknesses in the algorithms and characteristics of the thumbnailing process. Phash was found to be particularly poor when detecting thumbnails with fractals or repeated patterns (Figure 5.8), while Blockhash was poor when images possessed large areas with little to no variation in colour (Figure 5.9).

The weakness of each algorithm may be mitigated by using both simultaneously, and taking the lowest score from the thumbnail to the full-sized image. This effectively decreases the false negative rate to approximately 0.002%, or 1 in 50,000. However, even if it is assumed that there are multiple targets to detect in the thumbnail cache, the likelihood that contraband images would be missed using these hashing methods is perhaps too high in some contexts. However, as this method is designed for rapid triage, and already makes assumptions about images being in the thumbnail cache, this level of performance should be acceptable in most cases.

Unfortunately, the false negative rate cannot be reduced by simply increasing the distance threshold, as some thumbnails were observed to have a Hamming distance greater than 0.5 to the full-sized version, which is larger than the mean distance to a completely unrelated image. It is conceivable that there exists a perceptual hashing algorithm which performs better in this use case and would provide a higher degree of confidence that no thumbnail has been overlooked. An ideal algorithm should primarily address the problem of scale invariance, such that the same fingerprint is generated from an image regardless

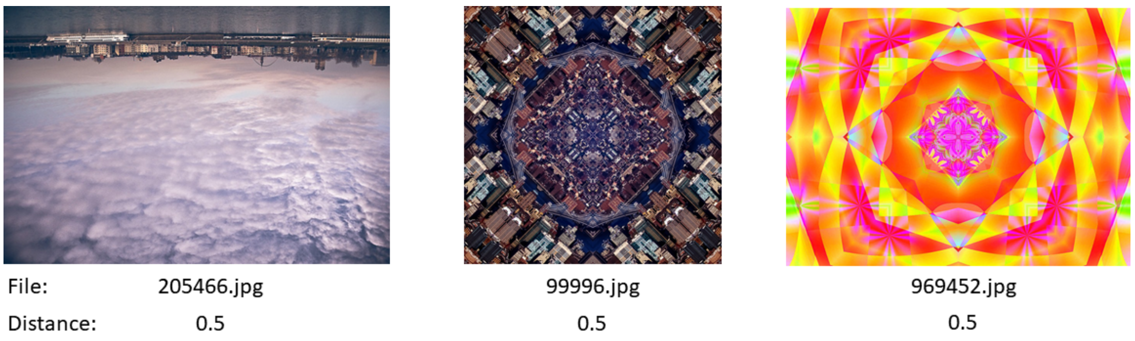


Fig. 5.8 Sample images where Phash performs poorly when comparing original image to 256px thumbnails. Distance is normalised Hamming distance.

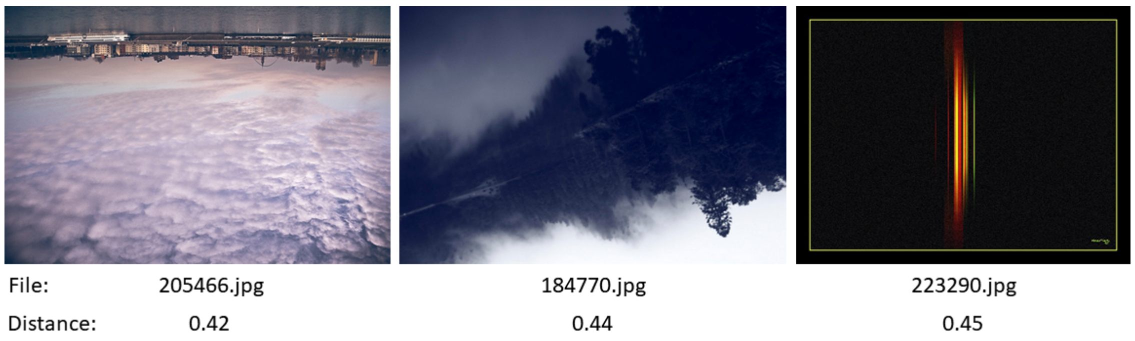


Fig. 5.9 Sample images where Blockhash performs poorly when comparing original image to 256px thumbnails. Distance is normalised Hamming distance.

of resizing. Ideally, all thumbnails would fall within some well defined distance of their full-sized counterparts, such that the false negative rate is effectively zero. However, a sufficiently low false negative rate may be tolerable, as long as it is unlikely to affect any decision making as a result of it [156]. One further consideration is the performance of looking up perceptual hashes in Hamming space, which is not as fast as the constant time lookups of traditional hashing mechanisms. However, this problem has solutions in the literature, such as multi-hash indexes [85].

No performance analysis was conducted for the perceptual hashing approach as the tested algorithms are not necessarily ideal for this use case. Since perceptual hashing techniques vary considerably, benchmarking these approaches may not be particularly informative. However, parsing the thumbnail cache is very fast, and perceptual hashes should be relatively inexpensive to extract from small thumbnail images. In practice, this means that for most perceptual hashing approaches, the main overhead is likely to be the complexity of looking up the hash in a contraband database.

5.7 Rapid Contraband Detection in the Cloud

The techniques described in this chapter can act as a form of disk level data reduction on local storage media, with a tiny fraction of a drive serving as a proxy for its content. These approaches may also be effective for cloud storage platforms, where thumbnails and item metadata are present. By greatly reducing the amount of data to read across the network, significant performance gains may be achieved, as with the sub-file hashing strategies in Section 4.5. This section first explores the possibility of using embedded metadata for contraband detection, much like the CRC checksums found in the Windows thumbnail cache, before discussing the possibility of using cloud storage thumbnails in lieu of full files. Both approaches assume that credentials have been obtained for accessing the cloud storage account, rather than relying on processing the local device cache [157].

5.7.1 Exploiting Embedded Discriminators

Cloud storage providers typically have a great deal of metadata associated with files and directories on their platform. This metadata allows for version tracking for client synchronisation, management of sharing/privacy properties, and provides the ability to uniquely identify an item on the platform. Checksums may also be used to verify the integrity of data once it has been transmitted across the network. As noted by Roussev et al. [109], this metadata is usually hidden, but APIs provided by the cloud service can access this information, which can be used to obtain a more complete acquisition of cloud storage.

The traditional approach to contraband detection is to acquire all of the binary data for a file, hash it, and then check if it exists in a database. However, when a highly discriminative identifier is already provided in file metadata, file data does not need to be fetched. As these discriminators are intended to track changes in a file, and are not able to be manipulated by users without changing file content, they are reliable data signatures. A list of cloud storage providers and their content signatures are provided below:

Google Cloud: CRC32C/MD5 – Google cloud provides both CRC32 checksums and MD5 hashes[39] for the purposes of verifying the integrity of downloaded files. CRC32C is standardised and uses a different polynomial to CRC32, while the standard MD5 hash is used for all non-composite objects on the platform.

Google Drive: MD5 – Files with binary data stored on Google drive have a property named md5checksum[40], which is a standard MD5 hash of the full file content.

[39]https://cloud.google.com/storage/docs/hashes-etags
[40]https://developers.google.com/drive/api/v3/reference/files

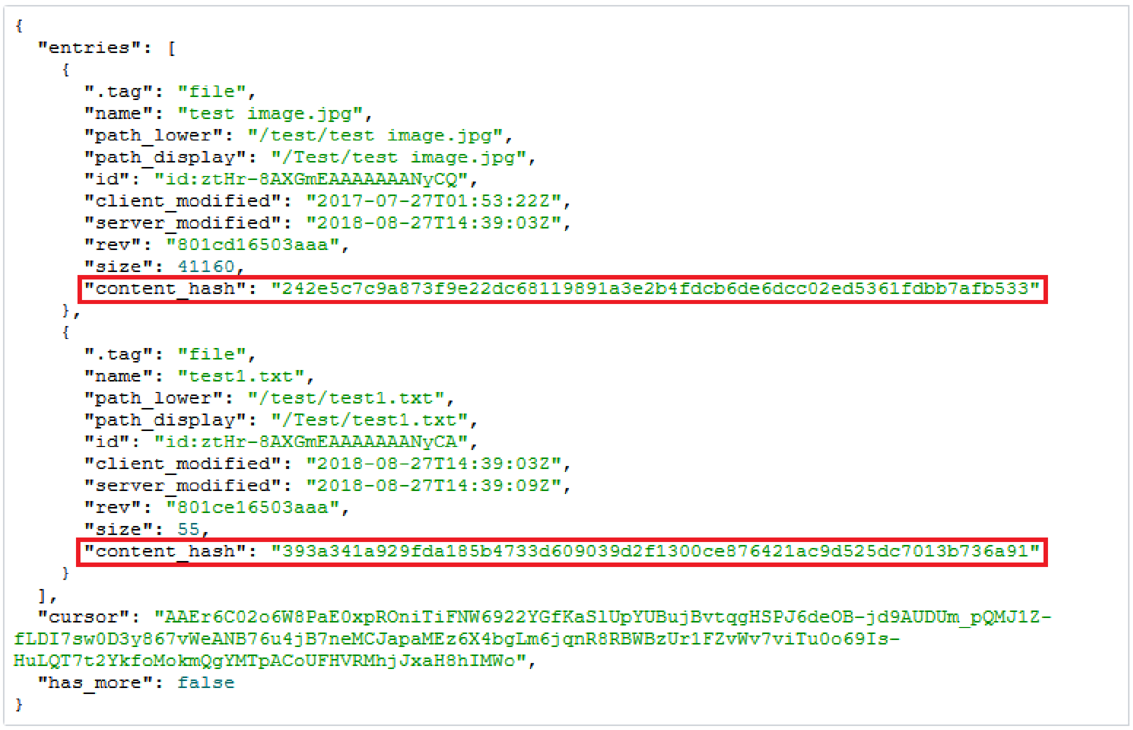
One Drive: CRC32/SHA1/XOR – Microsoft's One Drive makes use of three content hashes [41]: *i)* crc32Hash, a standard CRC32 checksum, *ii)* sha1Hash, a standard SHA1 full file hash, and *iii)* quickXorHash, a proprietary exclusive-OR (XOR) based hash with a fully documented implementation. CRC32 and SHA1 hashes are not available on OneDrive for Business, while quickXorHashes are not available on personal accounts.

Dropbox: content_hash – Dropbox uses a non-standard hashing approach for its content_hash file property[42], which is used for file verification. Files are broken into 4MiB blocks, which are hashed using the SHA256 algorithm. Block hashes are then concatenated into a single string, which is then hashed again with SHA256 to calculate the final hash.

All of the cloud platforms listed above make use of robust cryptographic hashes for at least one of their content signatures, meaning that it is not necessary to rely on CRC based checksums. These signatures can be exploited directly by an investigator without downloading any file content, and without calculating any hashes. All that is required is for a request to be made to the platform's API to obtain metadata for the files in question. This can typically be done in bulk, as with Dropbox, which returns content hashes for each file when requesting a directory listing, as depicted in Figure 5.10. Additionally, as most platforms use unmodified variants of hashing algorithms which are frequently used in forensics, existing contraband databases may be used without modification. The API response can be parsed for hash strings, which can then be compared directly with existing databases. However, in the case of Dropbox, a new contraband signature database would be required for this approach, as it is unlikely that an existing database has been generated using that particular hash concatenation method.

The metadata based approach is very fast, as it requires only strings to be requested across the network. Additionally, few API requests are needed on platforms such as Dropbox, as hashes for all files in each directory can be queried in a single request. This alleviates API rate limiting concerns, which may slow down the overall acquisition process if files are retrieved individually. However, the cloud platform hashes are as fragile as cryptographic hashes in disk based forensics, in that modifying a single bit in the file will generate a different hash. A simple obfuscation script could render contraband files invisible to cryptographic hash databases, such that a more robust approach may be used as a follow-up.

[41]https://docs.microsoft.com/en-us/onedrive/developer/rest-api/resources/hashes
[42]https://www.dropbox.com/developers/reference/content-hash



```
{
  "entries": [
    {
      ".tag": "file",
      "name": "test image.jpg",
      "path_lower": "/test/test image.jpg",
      "path_display": "/Test/test image.jpg",
      "id": "id:ztHr-8AXGmEAAAAAAANyCQ",
      "client_modified": "2017-07-27T01:53:22Z",
      "server_modified": "2018-08-27T14:39:03Z",
      "rev": "801cd16503aaa",
      "size": 41160,
      "content_hash": "242e5c7c9a873f9e22dc68119891a3e2b4fdcb6de6dcc02ed5361fdbb7afb533"
    },
    {
      ".tag": "file",
      "name": "test1.txt",
      "path_lower": "/test/test1.txt",
      "path_display": "/Test/test1.txt",
      "id": "id:ztHr-8AXGmEAAAAAAANyCA",
      "client_modified": "2018-08-27T14:39:03Z",
      "server_modified": "2018-08-27T14:39:09Z",
      "rev": "801ce16503aaa",
      "size": 55,
      "content_hash": "393a341a929fda185b4733d609039d2f1300ce876421ac9d525dc7013b736a91"
    }
  ],
  "cursor": "AAEr6C02o6W8PaE0xpROniTiFNW6922YGfKaSlUpYUBujBvtqgHSPJ6deOB-jd9AUDUm_pQMJ1Z-
fLDI7sw0D3y867vWeANB76u4jB7neMCJapaMEz6X4bgLm6jqnR8RBWBzUr1FZvWv7viTu0o69Is-
HuLQT7t2YkfoMokmQgYMTpACoUFHVRMhjJxaH8hIMWo",
  "has_more": false
}
```

Fig. 5.10 A screenshot of the Dropbox API V2 response to the list_folder endpoint. Content hashes (highlighted by boxes) are included in directory listings for easy bulk processing.

5.7.2 Processing Cloud Thumbnails

A robust method for detecting contraband on cloud services is to make use of the thumbnails provided on the platform. These thumbnails are generated for the client and web application previews, and can be accessed via the platform API. As thumbnails are a condensed form of the complete image content, they both reduce the amount of data to be transmitted across the network, and provide assurance about the content of the file. These thumbnails can then be cryptographically or perceptually hashed for comparison with contraband databases.

A small case study of the Dropbox platform was undertaken in order to understand the potential benefits of this approach. The largest 5000 images of the Flickr 1 Million dataset were used to create a subset for testing. This subset totals 1.57 GiB, with a mean file size of 337.15 KiB. Files were uploaded to the Dropbox platform and accessed via the same client workstation and 100Mbit Internet connection as the benchmarks in Section 4.5. Where possible, requests were made using the Dropbox Python SDK [158].

Four file access strategies were benchmarked for comparison:

Single File: Each file is downloaded individually, with a separate API request for each file using Dropbox.files_download.

Single Thumbnail: Thumbnails of size 128 were requested separately for each file using Dropbox.files_get_thumbnail.

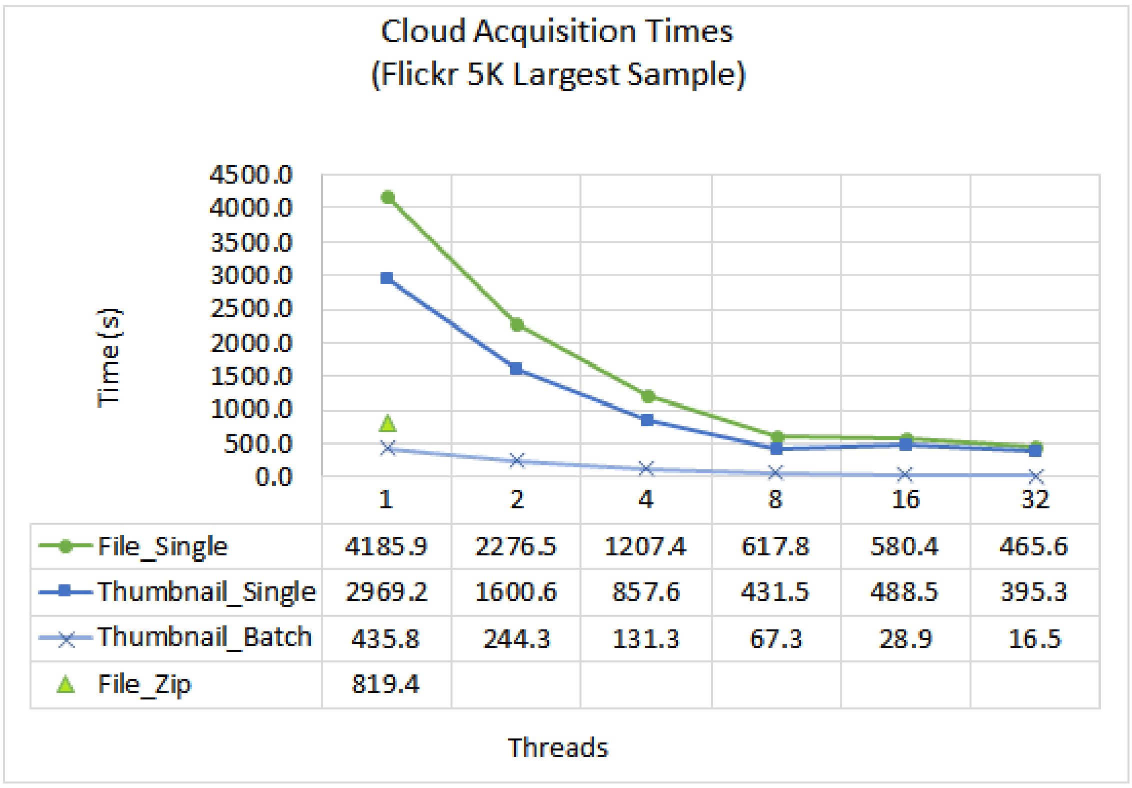
Batch Thumbnail: Thumbnails were downloaded in batches of 25, which is the maximum number of files allowed by Dropbox.files_get_thumbnail_batch.

Directory Zip: The entire directory was requested as a zip file via the files/download_zip HTTP endpoint using the Python 2.7 requests library [159]. This workaround was required as the Python SDK timed out when requesting the entire directory.

Each approach was repeated three times, with reported times representing the mean duration taken to acquire the data and read it into memory with no further processing. Multiple simultaneous requests were issued with up to 32 threads for all but the zip approach, which only makes a single request to the API.

When executing large numbers of API requests it is necessary to take note of any rate limiting functionality employed by the endpoint server. In this case, Dropbox does perform rate limiting but does not disclose metrics for when the limit is triggered. No rate limiting was observed in this experiment up to 32 threads, though a brief test with higher thread counts did result in some requests being rejected with a timeout. Code snippets for these benchmarks are provided in Appendix F.1.2 and F.1.3. In all cases except downloading the zip, files were first enumerated using Dropbox.files_list_folder, with this enumeration time being counted towards the overall time. Results for these benchmarks are provided in Figure 5.11.

Making many small requests to the Dropbox API is expensive, as each individual API request, regardless of size, comes with its own overheads. However, the Dropbox API has a very high rate limit and allows for many simultaneous requests over a reasonable period of time, facilitating reduced acquisition times at high thread counts. Requesting individual file thumbnails instead of full files resulted in a performance increase of 1.2–1.4×. As previously discussed in Section 3.6.3, file level data reduction approaches perform best when there is a good trade-off between the base access overhead and the cost of transporting file data. As thumbnail file sizes should be consistent, regardless of the size of the source file, it is expected that this performance gap would widen with larger files. This assumes that thumbnails of the requested size are already available on the platform and do not have to be generated as the request is made. Higher resolution images require more processing when carrying out re-scaling operations, which may mitigate some of the performance gains on larger files. However, as no decrease in request times was noted



	1	2	4	8	16	32
File_Single	4185.9	2276.5	1207.4	617.8	580.4	465.6
Thumbnail_Single	2969.2	1600.6	857.6	431.5	488.5	395.3
Thumbnail_Batch	435.8	244.3	131.3	67.3	28.9	16.5
File_Zip	819.4					

Fig. 5.11 Benchmark of mean time to acquire files from a Dropbox account. Comparison between downloading full files one at a time (File_Single) versus a single thumbnail at a time (Thumbnail_Single) and batch thumbnail downloading (Thumbnail_Batch), and downloading the directory as a zip file (File_Zip). Dataset is the largest 5000 files in Flickr 1 Million.

between runs, it is assumed that, at least for 128px thumbnails, they are already present on the Dropbox platform.

Acquiring thumbnails in batches of 25 reduced transfer times substantially. Between 1 and 8 threads the batch approach was approximately 9× faster than acquiring single files, reaching 20× at 16 threads, and 28× at 32 threads. This can be attributed to the 25-fold reduction in the number of API requests, and also to potential efficiency gains of the Dropbox server processing multiple related files simultaneously. This approach also compares favourably to downloading the entire directory via a zip file, which ends up being slower than the single file approaches at high thread counts. The reason for this is likely that zipping such a large directory requires much more memory and processing by the cloud provider, while the other approaches simply request that existing resources be transferred.

These results show that obtaining thumbnails, rather than full files, has the potential to decrease overall processing times of remote cloud storage. Small benefits are provided when requesting thumbnails one at a time on small Flickr files, while very large benefits are produced when accessing thumbnails in bulk on the Dropbox platform. However, much of the improvement can be attributed to reducing the number of API requests, which could also be achieved by batch downloading full files. Unfortunately, Dropbox does not provide a batch file download endpoint, such that a comparison cannot be made.

5.8 Thumbnail Triage in Practice

Thumbnail-based contraband detection is promising, and allows for rapid file content analysis. This section discusses practical concerns pertaining to this approach.

5.8.1 Thumbnail Robustness

Thumbnail-based approaches to contraband detection are afforded a slightly increased level of robustness over processing a full image. The reason for this is that when reducing the dimensions of an image, multiple pixels in the full image are inevitably represented by a single pixel in the thumbnail. This means that modifying individual pixels, in an attempt to foil full file cryptographic hashing, potentially results in no change to the thumbnail, as the differences are averaged out. However, this added robustness cannot necessarily be relied upon, as it may not take many modifications to change one or more pixels in the thumbnail. Additionally, whether or not a particular modification results in changes to thumbnail pixels is dependent on the particular thumbnailing algorithm, and how pixels are merged. Thumbnail comparisons, then, require a small degree of tolerance, but not necessarily complex perceptual hashing schemes when comparing thumbnails generated by the same algorithm. This robustness is lost when comparing images between platforms, or when the thumbnail generation approach can vary on a single platform.

5.8.2 Thumbnail Triage on Windows

The thumbnail cache on Windows operating systems can be used as a centralised catalogue of images on a device. However, it should be reiterated that there is no guarantee that an image which resides on a computer will be present in the thumbnail cache. The literature [31, 142] discusses various triggering mechanisms for caching, in some cases showing that images need not even be viewed to be present in the cache. Despite this, Windows does not appear to pro-actively cache all images on the device, instead waiting for some file/directory level triggering criteria. It should also be noted that if indexing is disabled for a particular directory or device then it will not generate thumbnails or Windows index entries, meaning that it is possible for items to be missing from the cache even if they have been viewed via explorer. Nonetheless, the thumbcache does appear to hold the most recently viewed thumbnails, such that it is, at minimum, a representation of the most recently previewed indexable, thumbnailable, items in Windows Explorer. The cache, then, can be seen as an up-to-date sample of images viewed on the device, and should be representative of which images have been accessed in normal use by a given user.

The user can attempt to remove traces of images and thumbnails from a device, affecting the sample present in the cache. Morris and Chivers [31] investigated deletion behaviour of thumbnails on Windows 7, finding that when the original image is deleted, the cache entry is not necessarily purged. This allows for the detection of contraband files which are no longer present in their full form. Similarly, thumbnails for removable media are also stored in the cache, regardless of whether or not the media is connected at the time. In practice, this extends the sample in the cache to recently used USB storage, networked drives, and recently deleted files. This can provide an indication that there are external storage media which are worth seeking out, further focusing the investigation and providing critical clues for the initial incident response.

However, the thumbnail cache itself can be cleared using the Disk Cleanup utility[43] or or the scheduled SilentCleanup version of the same process [147], which temporarily moves all thumbcache_*xxx*.db files to a subdirectory named ‘ThumbCacheToDelete’, with fresh databases taking their place [31]. At some point this temporary directory is deleted, though it is unclear how long it persists. This clean-up behaviour was also present in the experiments on Windows 10 for this chapter, and indicates that entire cache databases may be found in unallocated disk space, even after an attempt to purge them. As such, during triage it would be wise to check the thumbnail cache directory for deleted versions of ThumbCacheToDelete with a forensic preview tool.

Pseudocode for the thumbnail triage approach on Windows is provided in Algorithm 3. After mounting the device read-only, User directories on the device are enumerated, and the thumbcache_*xxx*.db files for each user are parsed in turn. Each thumbnail binary in the database file is then hashed and compared to the contraband hash set. If records in the database files match hashes for known contraband, the investigator can then make use of the thumbnail ID to locate the full-sized version of the image on the computer by referencing the Windows.edb database, which potentially contains the path to the full sized image. [161].

A final consideration when using traditional cryptographic hashes is that of memory limitations when running on low specification devices. Due to variations between Windows versions, and multiple thumbnail sizes provided by the cache, lookup databases would contain several hashes for the same source file. The inflation of the database may make

[43]And the more recent ‘Free up space now’ option provided Storage Sense on Windows 10 [160].

Algorithm 3: Thumbnail Triage - Pseudocode to count number of thumbnail database hits on a Windows machine.

```
Input: Evidence Drive
Output: Hit Count
hit_count = 0;
media = openReadOnly(disk);
for user in media/Users/ do
    thumbcache_dir = media/Users/user/Appdata/Local
                    /Microsoft/Windows/Explorer/;
    for thumbcache in thumbcache_dir do
        // thumcache_dir should also include ThumbCacheToDelete
        for record in thumbcache do
            thumbhash = hash(record.image);
            if thumbhash in contraband_db then
                hit_count++;
            end
        end
    end
end
return hit_count
```

it difficult to store in the main memory of many computers. A solution to this is to have targeted databases for different versions of Windows, or to reduce memory footprints using compressed data structures, such as bloom filters [45]. This is less of a concern for perceptual hash databases, as a single hash for each full-size image should be adequate for detecting thumbnails, irrespective of host platform and thumbnail dimensions.

Anti-forensics and Reliability

There are several limitations to this approach when performing triage on Windows. As the user can delete the cache, mark directories as non-indexable, or avoid using explorer to view files, thus avoiding caching, there is no guarantee that contraband will be represented in the cache. Additionally, not all items are cached even when viewed in explorer. If the user changes the file extension to a format which Windows does not create thumbnails for, or to an unknown extension, such as `.zzz`, then there will be no entry in the cache. Similarly, while Windows is fairly robust when generating previews for mismatched image formats (e.g. a JPEG renamed to .png will still be cached), giving a video extension, such as .mov, to an image file will cause it not to be cached, even if the video format supports caching. The user may also choose to disable caching altogether via group policy.

As a result, while the thumbnail cache is an excellent source of evidence on a device, it can be actively thwarted by a savvy user, and is therefore not 100% reliable as a standalone technique. However, the possibility of the user subverting the cache is outweighed by the potential utility and relatively inexpensive processing required to perform such lookups.

5.8.3 Thumbnail Triage on Other Operating Systems

While this work discusses centralised thumbnail caches for the Windows operating system, the literature indicates counterpart centralised caches on Ubuntu [143], Android [144], and MacOS [145], which could be similarly exploited. Ideally, perceptual hashes should be platform independent, and be directly comparable with any thumbnail cache. However, further work is required to explore the false positive and negative rates for thumbnails on non-Windows devices, as the thumbnail generation process may impact detection accuracy. Performance degradation will occur if a variety of aspect ratios are used when generating thumbnails, such as forcing squares or distorted representations of the original image when re-scaling. In the worst case, multiple perceptual hash databases can be maintained, selecting the most appropriate for a given platform. It is also recommended that hash databases include signatures for embedded EXIF thumbnails, as these may be used to create thumbnails, as with the 96 pixel thumbnails on Windows 7.

When employing traditional cryptographic hashing, the corresponding thumbnail generation APIs for each operating system must be utilised to generate the thumbnails for hashing. Failing this, another method for generating identical thumbnails on the platform must be discovered to automate the process. Ongoing maintenance is also required to verify that new operating system releases do not alter the method which is used to generate thumbnails. However, it is likely that related distributions of an operating system generate thumbnails in the same fashion. In this case, the labour is reduced, as, say, all Debian based Linux distributions may behave identically. On Linux, it is possible for third-party applications to generate thumbnails on a device, which would likely fall outside of the scope of cryptographic hash databases. This further emphasises the place of perceptual hashes as a generic solution to the problem of identifying contraband thumbnails.

5.8.4 Thumbnail Analysis on Cloud Platforms

Thumbnail analysis of a cloud platform behaves like the sub-file hashing approaches in earlier chapters, effecting file level data reduction. The author is not aware of any cloud storage platform which makes use of centralised cache stores which can be accessed efficiently. As such, thumbnails are utilised as reduced representations of individual files, with all files being processed in turn. This is fundamentally different from the effective disk sampling provided by analysing the thumbnail cache on Windows devices.

As cloud platforms already provide easily accessible file hashes (see Section 5.7.1), the role of thumbnail analysis is second-stage contraband detection. As with the process model in Breintinger et al. [52], multiple stages of contraband detection can be adopted. On the cloud, the first stage is to check hashes in the metadata. At this point, known, unmodified contraband should be detected. The second stage, thumbnail analysis, can be used to detect modified image variants through perceptual hash lookups, while reducing the network throughput bottleneck.

5.8.5 Thumbnail vs. Sub-file Triage

When using thumbnails on a per-file basis, as with cloud analysis, they have the same overall effect as sub-file detection strategies, reducing file level data reads. The effective data reduction achieved by thumbnails is slightly lower, with the mean thumbnails in this work corresponding to 26.2KiB for 96px and 15.5KiB for 256px thumbnails on Windows 10[44], which is still a tiny fraction of a reasonably sized image. Additionally, as thumbnails are direct representations of the full image content, they may be processed at the pixel level, which is not feasible with the sub-file approaches described in this thesis. In particular, heuristics such as flesh tone detection [162] may be employed to identify potentially unknown files of interest.

The real power of thumbnail processing is when it is applied to a centralised cache. This changes the data reduction from file level to disk level, and places thumbnail analysis in a new class of techniques. When utilised in this fashion, processing the thumbnail cache should be the first step in forensic triage, as it is incredibly fast. When processing the remainder of the disk, the data reduction methodology of Grier and Richard [43] can be used in combination with sub-file approaches to rapidly process the remainder of the disk. Both sub-file and thumbnail-based approaches, then, are not in direct competition, and have different benefits depending on the situation.

5.9 Conclusions

This chapter has shown that centralised thumbnail caches offer an opportunity to perform rapid forensic triage and contraband detection, potentially saving a huge number of investigator man hours. Cryptographic hash analysis of the Windows 10 thumbcache can be performed in a matter of seconds, even on low-end legacy equipment. The cache itself serves as a sample of recent images on the device, reducing processing of entire disks to a few hundred mebibytes.

[44]96 pixel thumbnails are bitmaps and generate larger files than the 256 pixel JPEGs. Figures calculated from the Flickr 1 Million thumbnails.

While this work focuses on the Windows 10 operating system, further research will allow the technique to be expanded to other operating systems and mobile devices. This can be achieved by further refining the application of perceptual hashing techniques to detect target thumbnails of any dimensions and compression ratio. Flexible thumbnail fingerprint matching will allow for generic signature databases to be maintained, which can be used to detect contraband swiftly across any device using centralised thumbnail stores. Finally, thumbnail-based approaches also offer file level data reduction for cloud investigations, particularly when they can be fetched in batches.

Chapter 6

Conclusions and Future Work

In order to tackle the volume problem in digital forensics, and begin to alleviate backlogs, this thesis states that: *reduced file representations may be used to mitigate forensic I/O and bandwidth bottlenecks, resulting in faster forensic processing, while maintaining forensic integrity*. Law enforcement backlogs are now measured in years, which has detrimental effects on the innocent and prevents justice from being delivered effectively to the guilty. The work in this thesis seeks to improve investigative turnaround and facilitate the eradication of such backlogs. This chapter discusses how each of the contributions supports the aims of the thesis, before discussing future work.

6.1 Overview of Contributions

Reduced file representations can potentially take many forms, but the primary goal in this thesis is always to reduce the amount of data required to identify an item of contraband, effecting faster processing. The work in Chapter 3 explores the possibility of generating highly distinct signatures from file type specific features, while reading as little of the file as possible. This approach requires the careful analysis of a file type specification, as well as an empirical study to determine which elements of the standard are actually used in the real world. Two file types were chosen for this work, PNG and JPEG, which are currently the most popular image formats for lossless and lossy compression, respectively. A different signature generation strategy was employed for each file type, with the PNG approach combining multiple low entropy features with a small chunk of image data, and the JPEG approach directly exploiting Huffman tables, which can be construed as a statistical image proxy. In both cases, signatures were found to be highly discriminative, with the recommendation that the PNG approach is used as part of a filtering scheme, requiring verification. Signatures were extracted using only 1–3% of the relatively small files used in testing, a percentage which will decrease with larger file sizes. These approaches were benchmarked against full file hashing to determine if they have the potential to reduce investigative processing times. It was discovered that accessing small chunks of each file is incredibly fast on solid-state media for any file size, while benefits on hard drives are only realised when files are not very small.

Generalising the partial file data reduction approach, Chapter 4 discusses sub-file hashing approaches which can be applied to any compressed file type. The most successful approach reads a fixed logical data block from the end of each file, resulting in perfect discrimination with 4096 byte blocks on a million file dataset. This approach has the benefit of not requiring in-depth knowledge of a particular file type, merely relying on the entropy provided by compressed data streams. Again, this method was shown to be particularly effective on solid-state media, with similar performance characteristics to techniques in the prior chapter. Sub-file hashing was also shown to perform well across both LAN and Internet connections, expanding the use case to networked and cloud environments. The work in both Chapters 3 and 4 indicate that partial files are an effective reduced file representation for detecting contraband, which also improves processing times significantly in most cases. For use in triage, this approach may be combined with file system metadata processing to sample the drive [43], allowing for a focused search. As modern images are typically much larger than the images tested in this work, greatly reduced processing time is expected in real-world scenarios, with projections showing improvements of up to two orders of magnitude. The evidence indicates that partial file approaches have the potential to reduce forensic backlogs, and directly supports the reduced file representation thesis.

An alternative approach to address the aims of this thesis is to make use of image thumbnails, which are reduced resolution representations of a full-sized image. Chapter 5 shows that the centralised thumbnail cache on Windows operating systems can be exploited directly for rapid disk triage. Thumbnail signatures can be generated using either cryptographic or perceptual hashing functions. When employing the former, due consideration must be given to the differences between operating systems, while the latter has the potential to utilise a single lookup database for any platform. Centralised thumbnail caches may be used as a proxy for content on an entire drive, with benchmarks showing that tens of thousands of thumbnails may be processed in a matter of seconds. Thumbnails are also commonly used on cloud storage services for client application previews. Experiments on the Dropbox platform showed that fetching individual thumbnails has some benefit over acquiring full files, but is most powerful when exploiting the batch request functionality, reducing API overheads. Overall, thumbnails are a robust stand-in for full file content, as they directly represent pixel content for a file. Thumbnail sizes are a tiny fraction of full file sizes, effecting an excellent level of data reduction. This method of mitigating I/O and network bandwidth is effective, particularly when dense thumbnail stores are easily accessible. When analysing local disk content, this approach may be used as first stage, near instantaneous, triage. Thumbnails may be used standalone, or may be augmented by disk level data reduction and the sub-file strategies from earlier chapters, for a more complete disk analysis. Thumbnail approaches are an alternative way of achieving reduced backlogs, directly addressing the aims of this thesis.

6.2 Achievement of Thesis Objectives

This thesis is focused on data reduction for the sake of reducing the quantity of data to be read from the base storage media, which in turn alleviates a portion of the forensic bottleneck attributed to the storage media bottleneck. To be useful in court, a forensic technique must also be robust to the point of being reliable as evidence. Further evaluation criteria were identified from the literature in Section 3.2, which elaborate on these fundamental requirements. The techniques in Chapters 3 and 4 had such requirements built-in by design. The work in Chapter 5 also has the same set of requirements, but as the focus was not on signature generation, instead choosing to use existing methods, some of the criteria are not directly applicable. What follows is a discussion of how the work in this thesis addresses the thesis statement and the expanded evaluation criteria from Section 3.2.

6.2.1 Uniqueness / Discriminating Power

High discriminative power is a necessary characteristic when performing automated contraband detection. Both the file type specific and generic sub-file approaches produced highly discriminative signatures. Optimised Huffman tables are essentially unique at the million image scale (Table 3.9), with false positives being generated by very similar images (Figure 3.17), which may actually be considered a strength over traditional cryptographic hashing. Indeed, this result is not surprising given that it was work in the field of Content Based Image Retrieval which inspired this method. The PNG approach shows that signatures can be built from lower entropy features, but that it works best on heterogeneous datasets. Despite this, even when the same encoder is used to generate the images 99.8% of images in the converted Govdocs dataset produced a unique signature (Figure 3.7). This effectively means a false positive rate of 1-in-500 images in the worst case, which is acceptable, but best paired with a more accurate technique to perform a verification check. The generic sub-file approach produces unique signatures when reading as little as 4KiB of data from the end of the file (Table 4.2). This is not surprising as the end of most compressed file types is simply a compressed, high entropy, data stream. This result should then hold for any file type which has a similar layout, and is very promising.

The thumbnail based approaches rely on both the discriminating power of the underlying hash and the characteristics of the thumbnailing process (Section 5.5.2). Very similar images may produce identical thumbnails by chance, but again this may be a positive feature. Cryptographic hashes of these images will produce unique signatures as long as the thumbnails are unique, while the perceptual hashing techniques used are somewhat at the mercy of the thumbnail re-scaling process (Figures 5.8 and 5.9). The false negative and positive rate can be controlled by selecting an appropriate distance threshold to determine if a thumbnail is a match or not, with a false positive rate of 1 in 10,000 being easily achievable (Table 5.2).

6.2.2 False Negatives and Robustness

Ideally, no illegal media should be missed when processing a disk for a forensic investigation. The risk of overlooking critical evidence has to be accepted when performing any kind of triage or data reduction but it is critical that this risk is minimised and quantifiable. Neither the file type specific (Chapter 3) or file agnostic (Chapter 4) sub-file signature generation approaches will produce false negatives when matching against the original file. This is possible because the signatures are deterministic, such that the same source file will always produce the same output signature, and all files are processed without selecting a sampling subset. In this respect, both sub-file approaches have the same level of assurance as traditional cryptographic hashing. However, the sub-file approaches are able to cope with some forms of binary or content based manipulation which cryptographic hashes cannot deal with. This was clear from specific examples (e.g. Figures 3.5, 3.17 and 3.18) in the dataset despite the lack of content or binary alterations to images in the dataset. Optimised Huffman tables will not change when the metadata is modified, but will change when the content of the image changes enough to alter the JPEG's DCT code frequencies, providing some level of flexibility. The PNG approach only targets metadata necessary to render the file, along with a small portion of binary data, meaning that the file has to be encoded differently, or the first few scanlines of the image would have to be modified to produce a negative result. Similarly, the Last 4KiB technique will only generate a different signature if the DCT codes or other encoding tables use to encode the image change, or if the changes to image content affect the last 4KiB of compressed scan data.

Thumbnails on a given operating system version are produced deterministically, such that no images will be missed using the cryptographic approach (Section 5.5.2). The weakness of particular perceptual hashing approaches may cause false negatives, with repeating patterns, fractals and solid background images causing problems with the tested algorithms (Figures 5.8 and 5.9). However, the risk of false negatives may be mitigated by combining techniques to avoid the shortcomings of both, in these experiments reducing the false negative rate to 1-in-50,000 (Section 5.6.2). When modifying an image the weakness of cryptographic hashing is somewhat mitigated by the re-scaling process, meaning that small pixel changes will likely have no effect in preventing the image from being detected, requiring more substantial changes across the image which will be reflected in the downscaled version. Perceptual hashing is built to tolerate such modifications, which, combined with the thumbnailing process, results in strong tolerance for content-based image modifications. The thumbnails are not affected by metadata or content preserving binary changes in the original image, meaning that only content-based attacks will have any effect on the lookup accuracy. One risk of false negatives when looking in the cache is that an image may simply not have a corresponding thumbnail (Section 5.8.2), however, it is argued that the speed of this approach outweighs the cost, making it always valuable, while it also provides a good sample of recently viewed images on the machine. Together this means that as long as the investigator has the appropriate expectations for this approach, the risk of missing evidence is not a great concern.

6.2.3 Generation Speed and Lookup Speed

Simply reducing the amount of data to be read from the storage media is not particularly effective unless the data can be processed as fast as it is acquired [11]. The timed benchmarks in this thesis combine figures for the time to read the data, extract the signature, and perform a lookup of the signature in a database. The complexity of the lookups in all benchmarks, for sub-file and thumbnails alike, was constant time, O(1), as they were simple string lookups in a Python dictionary or C++ unordered map. The benchmark times, then, effectively communicate the time to read and generate signatures, which are then compared directly to the full file hashing approach in Chapters 3 and 4. As all sub-file approaches read approximately the same amount of data, with the exception of the larger Last *n* values, they all perform very similarly (Tables 4.3, 4.4, and Figure 4.6), with CPU bottlenecks not being a constraint in the experiments (Figure C.3). The effective speed improvement over full file cryptographic hashing is up to 70× for 500KiB files on SSDs, and 3× for the same files on an HDD (Tables 4.3 and 4.4). This shows that these techniques benefit greatly from the reduced amount of data read from the underlying storage media, while also being robust, as per Section 6.2.2.

The thumbnail approach can make use of traditional cryptographic hashing and is therefore not directly comparable to a baseline approach. However, there is no approach in the literature which claims to identify contraband on an HDD in under a minute, regardless of disk size. Cryptographic hashes are fast to calculate, meaning that the main concern is the execution and lookup speed of perceptual hashes for a unified lookup database. Such concerns have been documented in the literature [52], showing that while perceptual hashes are relatively slow compared to other hashing methods, they are not slow enough to invalidate the approach. Additionally, as the thumbnails in the cache are already downscaled the processing time of the perceptual hashes is going to be less onerous. Database lookup strategies are more of a concern, but there are also strategies in the literature to avoid millions of pairwise comparisons [85].

6.2.4 Compression/Signature Length

The length of the signature is important when considering possible memory constraints at scale. Contraband databases can contain millions of items, which should ideally be stored in RAM for optimal lookup speeds [45]. Cryptographic and perceptual hashing techniques typically have fixed length digests as their output, while bytewise matching schemes are often variable length [52]. The PNG signature generation method in Chapter 3 is a variable length as it depends on the size of the features provided in the `IHDR` chunk, but was always observed to be smaller than 64 bytes, which is less than the digest length of SHA512 (Section 3.4.3). Optimised Huffman tables for JPEGs were observed to be typically less than 300 bytes with no compression, which would be equivalent in length to a 2400 bit cryptographic hash (Section 3.5.4). This is perhaps too large to store natively, but a compression technique or cryptographic digest may be generated from the Huffman signature in order to produce a fixed, or compressed, signature for better memory utilisation. The sub-file signatures from Chapter 4 use cryptographic hashes and are therefore a fixed size, with the length depending on the chosen hashing algorithm. Similarly, thumbnail signatures in Chapter 5 make use of existing cryptographic and perceptual hashing schemes, such that they will also be fixed length. In no case does the size of the signature scale with the size of the file, with compression being possible to reduce memory overhead. As such, all three high-level approaches in this work are suitable for in-memory databases for contraband detection.

6.2.5 Anti-forensics for Sub-file approaches

Signature based contraband detection is limited to locating identical, or similar, files on a device, depending on the approach, and cannot find completely new files of interest. In the case of traditional cryptographic hashes, flipping a single bit in the file completely changes the hash, making them trivial to defeat with small file modifications [51]. Perceptual hashes are more robust and can tolerate both binary and content level manipulations [40, 52], but have their own weaknesses (Sections 2.3.1 and 2.3.2, Figures 5.8 and 5.9). The approach in Chapters 3 and 4 reduce the risk of small file manipulations throwing off the signature by focusing on the content, or encoding properties, of the image. As such, an attacker would have to modify how the image is encoded, or change the content itself, in some way in order to affect these signatures. However, as discussed above in Section 6.2.2, the sub-file approaches are more robust than traditional cryptographic hashing. A wider range of attacks can be used against the thumbnail cache. As discussed in Section 5.8.2, disabling indexing or making changes to group policies, or changing file extensions, can cause thumbnails to never be cached for a file. Additionally, the caches themselves may be deleted using built-in functionality of the Windows OS. A user could therefore make thumbcache analysis impossible, preventing this form of triage. However, the perceptual hashing and downscaling components of the thumbnail triage process make it less susceptible to image modifications, or even re-encoded images, such that it is an effective option when available.

When encryption is used to obfuscate the original files, the sub-file approaches in Chapters 3 and 4 will not be able to detect contraband items unless they are decrypted first. Thumbnail triage, however, will still be of use in this case as long as the operating system partition is accessible (i.e. not encrypted), as thumbnails are generated for images when they are mounted by the Windows OS, regardless of whether or not the original volume or container was encrypted[45].

6.2.6 Evaluation of Thesis Statement

All of the techniques in this thesis fulfill the relevant criteria, providing fast mechanisms to process forensic evidence while making little or no compromise in the reliability of evidence detection. Reduced representations often affect large speed increases when compared to full file cryptographic hashing, and are capable of reducing processing times by two orders of magnitude given large enough files.

6.3 Implications for Digital Forensics

In a world where storage capacities and the ability for the average user to generate data are constantly increasing, forensic examiners require efficient tools to avoid being overwhelmed. This thesis provides effective approaches for dealing with data volume in forensics, without requiring vast computational resources, as with approaches in prior work [11]. One important distinction between the data reduction approaches in this work compared to the existing literature is that the reduction is deterministic, with the "sampling" equivalent being done at the logical level for each file, as opposed to the disk level. Prior work focuses on randomly selecting low-level disk blocks [29, 45], which

[45]That is, the volume or container must be decrypted to be mounted, meaning that the files are in plaintext and are able to be cached.

may miss critical evidence in the sampling process, and is also susceptible to block level manipulations which render the hashes incapable of detecting contraband. In contrast, the sub-file approaches in this work are more robust to file level manipulation, but also process all files such that files with the matching signature will always be detected. However, one downside to the logical sub-file approach is the need for a file system to be present and intact, while random block sampling can be used on a raw disk with a damaged file system.

Both Grier and Richard [43] and Dalins et al. [163] make use of data reduction schemes which rely on heuristics to parse the file system for the most likely sources of information. These solutions have the same need for an intact file system, but also effectively sample from the disk, albeit it in a deterministic fashion. Again, this sampling may miss critical data, particularly if the case type heuristics overlook unusual evidence. However, these heuristic samples complement the work in this thesis very well, as they can be used as a starting point for processing. This thesis does not propose a method for selecting the order in which the file system should be processed, instead focusing on making the processing of individual files faster. The above work on file system data reduction can be used to strategically select which parts of the file system to scan first, potentially reducing the initial time taken to detect the first pieces of contraband on a device.

A strategic approach to evidence collection and processing will be necessary to keep pace with developments in storage technology. This thesis provides a promising starting point for non-mechanical media, with random access, sub-file techniques performing very well on SSDs, with expected performance to be higher on Intel Optane based devices. However, the expansion of the storage capacity of mechanical media is also expected, largely thanks to the HAMR technology [164] which applies heat to the read head to effect higher areal density on a device. The road map for the technology suggests that 100TB drives will be available by 2025 [165], which may be disastrous for the field of digital forensics unless new techniques are adopted. Based on the results of this thesis, it seems prudent to begin contraband image investigations with analysis of the thumbnail cache to detect easily accessible low hanging fruit. This is particularly effective as cache does not scale in size with the disk. A strategic walk of the file system can then be used to target sections of the disk for analyses, which can then utilise sub-file techniques for reduced processing times. Indeed, as the HAMR technology is accompanied by a multi-actuator technology (MACH.2) [166], a random sampling of the entire disk could also be conducted at the same time as a focused sub-file approach, effecting a double-pronged triage approach. It may be necessary to make trade-offs between absolute completeness and the requirement to process devices quickly if the field is to keep pace with such technological developments.

6.4 Future Work

Sub-file signature approaches have been shown to be effective for multiple image formats, however, testing has been largely limited to files of the PNG and JPEG format. While these approaches should be effective on all media types which are relevant to contraband detection, future work could expand on this thesis to empirically verify this. Of particular interest are storage formats which make use of low-resolution image scans early in the file, as with progressive JPEGs. These scans could be exploited as a proxy for the entire file, essentially serving as a thumbnail for contraband detection.

Additional storage technologies could also be explored, however, the analysis of effective read speeds in Section 3.6.3 provides a good basis for reasoning about performance of untested media types, such as Intel Optane devices. Instead, a more useful endeavour would be to explore alternative approaches to forensics which make the best use of non-mechanical storage. Much of the research in the field has assumed that the underlying storage technology is a hard drive. However as more devices make use of flash-based media, the community should explore the opportunities afforded by the different performance characteristics of such devices, as with the work in Chapters 3 and 4. Each new storage technology provides an opportunity to re-evaluate best practice and optimise forensic processes.

Finally, triage approaches utilising centralised thumbnail caches may be further developed. In order to generalise this approach, and provide a high-level of robustness and tolerance to varied thumbnail processes, more work is required to identify applicable perceptual hashing techniques. A perceptual hashing algorithm which is tailored for the comparison of full-sized images and their thumbnails would allow for initial triage to be performed in seconds on any device using a modern operating system. This would require thumbnail datasets to be collected from a wide variety of platforms currently in use. These datasets can then be used to evaluate existing perceptual hashes to determine which have the fewest false positives and negatives overall, or to identify a variety of complementing approaches. This may also require the development of a new perceptual hashing approach, which has downscaling invariance, and also accommodates slightly shifted aspect ratios and thumbnail aspects. A highly tolerant perceptual hash would allow desktop computers, laptops, and mobile devices containing obvious contraband to be detected rapidly, providing investigators with immediate, actionable, information.

References

[1] McKeown S, Russell G, Leimich P. Fast Filtering of Known PNG Files Using Early File Features. In: Annual ADFSL Conference on Digital Forensics, Security and Law. Daytona Beach, Florida, USA: ADFSL; 2017. Available from: https://commons.erau.edu/adfsl/2017/papers/1/.

[2] McKeown S, Russell G, Leimich P. Fingerprinting JPEGs With Optimised Huffman Tables. Journal of Digital Forensics, Security and Law. 2018;.

[3] McKeown S, Russell G, Leimich P. Sub-file Hashing Strategies for Fast Contraband Detection. In: IEEE International Conference on Cyber Security and Protection of Digital Services (Cyber Security 2018). Glasgow, UK: IEEE; 2018. .

[4] McKeown S, Russell G, Leimich P. Reducing the Impact of Network Bottlenecks on Remote Contraband Detection. In: IEEE International Conference on Cyber Security and Protection of Digital Services (Cyber Security 2018). Glasgow, UK: IEEE; 2018. .

[5] Data Never Sleeps 6 | Domo; 2018. Available from: https://www.domo.com/learn/data-never-sleeps-6.

[6] Big Data Solutions | IBM;. Available from: https://www.ibm.com/it-infrastructure/solutions/big-data.

[7] Garfinkel SL. Digital forensics research: The next 10 years. Digital Investigation. 2010 Aug;7:S64–S73. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287610000368.

[8] Beebe N, Clark J. Dealing with Terabyte Data Sets in Digital Investigations. In: Pollitt M, Shenoi S, editors. Advances in Digital Forensics. vol. 194 of IFIP - The International Federation for Information Processing. Springer US; 2005. p. 3–16. Available from: http://dx.doi.org/10.1007/0-387-31163-7_1.

[9] Marchon B, Pitchford T, Hsia YT, Gangopadhyay S. The head-disk interface roadmap to an areal density of Tbit/in2. Advances in Tribology. 2013;2013:1–8. Available from: http://www.hindawi.com/journals/at/2013/521086/.

[10] Schmid P. Hard Drives: 40 MB To 750 GB - 3,500 To 10,000 RPM - 15 Years Of Hard Drive History: Capacities Outran Performance; 2006. Available from: https://www.tomshardware.com/reviews/15-years-of-hard-drive-history,1368-2.html.

[11] Roussev V, Quates C, Martell R. Real-time digital forensics and triage. Digital Investigation. 2013 Sep;10(2):158–167. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287613000091.

[12] Gasior G. WD's Red 4TB hard drive reviewed; 2013. Available from: https://techreport.com/review/25391/wd-red-4tb-hard-drive-reviewed/4.

[13] Biggs S, Vidalis S. Cloud Computing: The impact on digital forensic investigations. IEEE; 2009. p. 1–6. Available from: http://ieeexplore.ieee.org/document/5402561/.

[14] Federal Bureau of investigation (FBI). Regional Computer Forensics Laboratory Program 2013 Annual Report. Regional Computer Forensics Laboratory; 2013. Available from: https://www.rcfl.gov/downloads/documents/fiscal-year-2013.

[15] Quick D, Choo KKR. Impacts of increasing volume of digital forensic data: A survey and future research challenges. Digital Investigation. 2014 Dec;11(4):273–294. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287614001066.

[16] Franqueira VNL, Bryce J, Al Mutawa N, Marrington A. Investigation of Indecent Images of Children cases: Challenges and suggestions collected from the trenches. Digital Investigation. 2018 Mar;24:95–105. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287617302669.

[17] Goldberg A. Child abuse cases delayed by police backlog; 2015. Available from: http://www.bbc.co.uk/news/uk-34713745.

[18] Lillis D, Becker B, O'Sullivan T, Scanlon M. Current Challenges and Future Research Areas for Digital Forensic Investigation. In: Annual ADFSL Conference on Digital Forensics, Security and Law. Daytona Beach, Florida, USA; 2016. ArXiv: 1604.03850. Available from: https://commons.erau.edu/adfsl/2016/tuesday/6/.

[19] Casey E, Ferraro M, Nguyen L. Investigation Delayed Is Justice Denied: Proposals for Expediting Forensic Examinations of Digital Evidence. Journal of Forensic Sciences. 2009 Nov;54(6):1353–1364. Available from: http://doi.wiley.com/10.1111/j.1556-4029.2009.01150.x.

[20] Hoffer TA, Shelton JLE, Behnke S, Erdberg P. Exploring the Impact of Child Sex Offender Suicide. Journal of Family Violence. 2010 Nov;25(8):777–786. Available from: http://link.springer.com/10.1007/s10896-010-9335-3.

[21] Edmundson D, Schaefer G. An overview and evaluation of JPEG compressed domain retrieval techniques. In: ELMAR, 2012 Proceedings; 2012. p. 75–78.

[22] Schaefer G, Edmundson D, Takada K, Tsuruta S, Sakurai Y. Effective and Efficient Filtering of Retrieved Images Based on JPEG Header Information. IEEE; 2012. p. 644–649. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6395154.

[23] Edmundson D, Schaefer G. Fast JPEG image retrieval using optimised Huffman tables. In: Pattern Recognition (ICPR), 2012 21st International Conference on. IEEE; 2012. p. 3188–3191. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6460842.

[24] Edmundson D, Schaefer G. Very Fast Image Retrieval Based on JPEG Huffman Tables. In: 2013 2nd IAPR Asian Conference on Pattern Recognition (ACPR); 2013. p. 29–33.

[25] Farid H. Digital image ballistics from JPEG quantization. Technical Report TR2006-583, Department of Computer Science, Dartmouth College; 2006.

[26] Kee E, Johnson MK, Farid H. Digital image authentication from JPEG headers. Information Forensics and Security, IEEE Transactions on. 2011;6(3):1066–1075. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5732683.

[27] Gloe T. Forensic analysis of ordered data structures on the example of JPEG files. In: Information Forensics and Security (WIFS), 2012 IEEE International Workshop on. IEEE; 2012. p. 139–144. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6412639.

[28] Garfinkel S, Nelson A, White D, Roussev V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. Digital Investigation. 2010 Aug;7:S13–S23. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287610000307.

[29] Young J, Foster K, Garfinkel S, Fairbanks K. Distinct sector hashes for target file detection. Computer. 2012;45(12):28–35. Available from: http://ieeexplore.ieee.org/abstract/document/6311397/.

[30] Quick D, Choo KKR. Data reduction and data mining framework for digital forensic evidence: storage, intelligence, review and archive. 2014;Available from: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2497796.

[31] Morris S, Chivers H. An analysis of the structure and behaviour of the Windows 7 operating system thumbnail cache. In: Proceedings from 1st Cyberforensics Conference; 2011. Available from: https://www.researchgate.net/profile/Sarah_Morris7/publication/262327134_An_analysis_of_the_structure_and_behaviour_of_the_Windows_7_operating_system_thumbnail_cache/links/00b7d5374e0aa2e5d8000000.pdf.

[32] Morris SLA. An investigation into the identification, reconstruction, and evidential value of thumbnail cache file fragments in unallocated space. 2013;Available from: https://dspace.lib.cranfield.ac.uk/handle/1826/8034.

[33] Pollitt MM. Triage: A practical solution or admission of failure. Digital Investigation. 2013 Sep;10(2):87–88. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287613000030.

[34] Shaw A, Browne A. A practical and robust approach to coping with large volumes of data submitted for digital forensic examination. Digital Investigation. 2013 Sep;10(2):116–128. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287613000327.

[35] Harichandran VS, Breitinger F, Baggili I, Marrington A. A cyber forensics needs analysis survey: Revisiting the domain's needs a decade later. Computers & Security. 2016 Mar;57:1–13. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0167404815001595.

[36] Hitchcock B, Le-Khac NA, Scanlon M. Tiered forensic methodology model for Digital Field Triage by non-digital evidence specialists. Digital Investigation. 2016 Mar;16:S75–S85. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287616300044.

[37] Carroll OL, Brannon SK, Song T. Computer Forensics: Digital Forensic Analysis Methodology. 2008;p. 9.

[38] Roussev V, Quates C. Content triage with similarity digests: The M57 case study. Digital Investigation. 2012 Aug;9:S60–S68. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287612000370.

[39] Garfinkel SL. Digital media triage with bulk data analysis and bulk_extractor. Computers & Security. 2013 Feb;32:56–72. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0167404812001472.

[40] Hadmi A, Ouahman AA, Said BAE, Puech W. Perceptual image hashing. INTECH Open Access Publisher; 2012. Available from: http://cdn.intechopen.com/pdfs/36921/InTech-Perceptual_image_hashing.pdf.

[41] Quick D, Choo KKR. Big forensic data reduction: digital forensic images and electronic evidence. Cluster Computing. 2016 Jun;19(2):723–740. Available from: http://link.springer.com/10.1007/s10586-016-0553-1.

[42] Magnet IEF;. Available from: https://www.magnetforensics.com/magnet-ief/.

[43] Grier J, Richard GG. Rapid forensic imaging of large disks with sifting collectors. Digital Investigation. 2015 Aug;14:S34–S44. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287615000511.

[44] Garfinkel SL, McCarrin M. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. Digital Investigation. 2015 Aug;14, Supplement 1:S95–S105. Available from: http://www.sciencedirect.com/science/article/pii/S1742287615000468.

[45] Penrose P, Buchanan WJ, Macfarlane R. Fast contraband detection in large capacity disk drives. Digital Investigation. 2015 Mar;12, Supplement 1:S22–S29. Available from: http://www.sciencedirect.com/science/article/pii/S1742287615000080.

[46] Bharadwaj NK, Singh U. Efficiently searching target data traces in storage devices with region based random sector sampling approach. Digital Investigation. 2018 Mar;Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287617303249.

[47] Sanford DP, Protection C, Group AIW, others. The Child Abuse Image Database (CAID). 2015;.

[48] Foundation IW. Image Hash List;. Available from: /our-services/image-hash-list.

[49] Rivest R. The MD5 message-digest algorithm. RFC Editor. 1992;Available from: http://tools.ietf.org/html/rfc1321.

[50] Gallagher P, Director A. Secure hash standard (shs). FIPS PUB. 2008;p. 180–3. Available from: http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[51] Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digital Investigation. 2006 Sep;3, Supplement:91–97. Available from: http://www.sciencedirect.com/science/article/pii/S1742287606000764.

[52] Breitinger F, Liu H, Winter C, Baier H, Rybalchenko A, Steinebach M. Towards a process model for hash functions in digital forensics. In: International Conference on Digital Forensics and Cyber Crime. Springer; 2013. p. 170–186.

[53] Roussev V, Chen Y, Bourg T, Richard GG. md5bloom: Forensic filesystem hashing revisited. Digital Investigation. 2006 Sep;3:82–90. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287606000740.

[54] Fowler G, Noll LC, Vo KP, Eastlake D, Hansen T. The FNV non-cryptographic hash algorithm. Ietf-draft. 2011;.

[55] Roussev V. Data fingerprinting with similarity digests. In: IFIP International Conference on Digital Forensics. Springer; 2010. p. 207–226.

[56] Roussev V. An evaluation of forensic similarity hashes. Digital Investigation. 2011 Aug;8:S34–S41. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287611000296.

[57] Oliver J, Forman S, Cheng C. Using Randomization to Attack Similarity Digests. In: International Conference on Applications and Techniques in Information Security. Springer; 2014. p. 199–210.

[58] Baier H, Breitinger F. Security aspects of piecewise hashing in computer forensics. In: IT Security Incident Management and IT Forensics (IMF), 2011 Sixth International Conference on. IEEE; 2011. p. 21–36. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5931110.

[59] Breitinger F, Baier H, Beckingham J. Security and implementation analysis of the similarity digest sdhash. In: First International Baltic Conference on Network Security & Forensics (NeSeFo); 2012. Available from: http://www.dasec.h-da.de/wp-content/uploads/2012/08/2012_08_Breitinger_NeSeFo.pdf.

[60] Breitinger F, Baier H. Properties of a similarity preserving hash function and their realization in sdhash. In: ISSA; 2012. p. 1–8. Available from: http://icsa.cs.up.ac.za/issa/2012/Proceedings/Full/43_Paper.pdf.

[61] Chang D, Sanadhya SK, Singh M, Verma R. A Collision Attack On Sdhash Similarity Hashing. In: Proceedings of 10th Intl. Conference on Systematic Approaches to Digital Forensic Engineering; 2015. Available from: http://sadfe2015.safesocietylabs.com/wp-content/uploads/2015/10/A-Collision-Attack-on-Sdhash-Similarity-Hashing.pdf.

[62] Martínez VG, Álvarez FH, Encinas LH. State of the art in similarity preserving hashing functions. In: Proceedings of the International Conference on Security and Management (SAM). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp); 2014. p. 1. Available from: http://search.proquest.com/openview/4289c772d85d64eb55a50a764f79b3d8/1?pq-origsite=gscholar.

[63] Roussev V, Richard GG, Marziale L. Multi-resolution similarity hashing. Digital Investigation. 2007 Sep;4:105–113. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287607000473.

[64] Breitinger F, Baier H. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In: International Conference on Digital Forensics and Cyber Crime. Springer; 2012. p. 167–182. Available from: http://link.springer.com/chapter/10.1007/978-3-642-39891-9_11.

[65] Winter C, Schneider M, Yannikos Y. F2S2: Fast forensic similarity search through indexing piecewise hash signatures. Digital Investigation. 2013 Dec;10(4):361–371. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287613000789.

[66] Breitinger F, Baier H, White D. On the database lookup problem of approximate matching. Digital Investigation. 2014 May;11:S1–S9. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287614000061.

[67] David JP. Max-hashing fragments for large data sets detection. In: 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig). IEEE; 2013. p. 1–6. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6732307.

[68] Allen TA. Non-RDS Hash Sets; 2016. Available from: https://www.nist.gov/itl/ssd/cs/non-rds-hash-sets.

[69] Sadowski C, Levin G. Simhash: Hash-based similarity detection. Citeseer; 2007.

[70] Broder AZ. On the resemblance and containment of documents. In: Compression and Complexity of Sequences 1997. Proceedings. IEEE; 1997. p. 21–29. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=666900.

[71] Breitinger F, Astebøl KP, Baier H, Busch C. mvHash-B-a new approach for similarity preserving hashing. In: IT Security Incident Management and IT Forensics (IMF), 2013 Seventh International Conference on. IEEE; 2013. p. 33–44.

[72] Breitinger F, Baier H. A fuzzy hashing approach based on random sequences and hamming distance. In: Proceedings of the Conference on Digital Forensics, Security and Law. Association of Digital Forensics, Security and Law; 2012. p. 89. Available from: http://search.proquest.com/openview/28cc97ad51028de51e1de35bfe5efa7d/1?pq-origsite=gscholar.

[73] Rafiee G, Dlay SS, Woo WL. A review of content-based image retrieval. In: Communication Systems Networks and Digital Signal Processing (CSNDSP), 2010 7th International Symposium on. IEEE; 2010. p. 775–779. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5580313.

[74] Swain MJ, Ballard DH. Color indexing. International journal of computer vision. 1991;7(1):11–32. Available from: http://link.springer.com/article/10.1007/BF00130487.

[75] Manjunath BS, Ohm JR, Vasudevan VV, Yamada A. Color and texture descriptors. IEEE Transactions on circuits and systems for video technology. 2001;11(6):703–715. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=927424.

[76] Venkatesan R, Koon SM, Jakubowski MH, Moulin P. Robust image hashing. In: Image Processing, 2000. Proceedings. 2000 International Conference on. vol. 3. IEEE; 2000. p. 664–666. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=899541.

[77] Wallace GK. The JPEG still picture compression standard. Consumer Electronics, IEEE Transactions on. 1992;38(1):xviii–xxxiv. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=125072.

[78] Fridrich J. Robust bit extraction from images. In: Multimedia Computing and Systems, 1999. IEEE International Conference on. vol. 2; 1999. p. 536–540 vol.2.

[79] Fridrich J, Goljan M. Robust hash functions for digital watermarking. In: Information Technology: Coding and Computing, 2000. Proceedings. International Conference on. IEEE; 2000. p. 178–183. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=844203.

[80] Steinebach M. Robust hashing for efficient forensic analysis of image sets. In: International Conference on Digital Forensics and Cyber Crime. Springer; 2011. p. 180–187. Available from: http://link.springer.com/chapter/10.1007/978-3-642-35515-8_15.

[81] Standaert FX, Lefebvre E, Rouvroy G, Macq B, Quisquater JJ, Legat JD. Practical evaluation of a radial soft hash algorithm. In: International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II. vol. 2. IEEE; 2005. p. 89–94. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1425127.

[82] Ith T. Microsoft's PhotoDNA: Protecting Children and Businesses in the Cloud. Retrieved from Microsoft News Center: https://news. microsoft. com/features/microsoftsphotodna-protecting-children-and-businesses-in-the-cloud; 2015.

[83] Aldhous P. The digital search for victims of child pornography. New Scientist. 2011;210(2807):23 – 24. Available from: http://www.sciencedirect.com/science/article/pii/S0262407911607914.

[84] Foster K. Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus. Monterey, California. Naval Postgraduate School; 2012. Available from: http://calhoun.nps.edu/handle/10945/17365.

[85] Norouzi M, Punjani A, Fleet DJ. Fast search in hamming space with multi-index hashing. In: Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on. IEEE; 2012. p. 3108–3115. Available from: http://ieeexplore.ieee.org/abstract/document/6248043/.

[86] Piva A. An Overview on Image Forensics. ISRN Signal Processing. 2013;2013:1–22. Available from: http://www.hindawi.com/journals/isrn.signal.processing/2013/496701/.

[87] Gloe T, Kirchner M, Winkler A, Böhme R. Can we trust digital image forensics? In: Proceedings of the 15th international conference on Multimedia. ACM; 2007. p. 78–86. Available from: http://dl.acm.org/citation.cfm?id=1291252.

[88] Farid H. Digital image ballistics from JPEG quantization: A followup study. Department of Computer Science, Dartmouth College, Tech Rep TR2008-638. 2008;Available from: http://www.cs.dartmouth.edu/farid/downloads/publications/tr08.pdf.

[89] Kornblum JD. Using JPEG quantization tables to identify imagery processed by software. Digital Investigation. 2008 Sep;5:S21–S25. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287608000285.

[90] Mahdian B, Saic S, Nedbal R. JPEG quantization tables forensics: a statistical approach. In: Computational Forensics. Springer; 2010. p. 150–159.

[91] He P, Wen X, Zheng W. A Simple Method for Filtering Image Spam. IEEE; 2009. p. 910–913. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5223148.

[92] Biggio B, Fumera G, Pillai I, Roli F. A survey and experimental evaluation of image spam filtering techniques. Pattern Recognition Letters. 2011 Jul;32(10):1436–1446. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0167865511000936.

[93] Krasser S, Tang Y, Gould J, Alperovitch D, Judge P. Identifying image spam based on header and file properties using C4. 5 decision trees and support vector machine learning. In: Information Assurance and Security Workshop, 2007. IAW'07. IEEE SMC. IEEE; 2007. p. 255–261.

[94] Cortes C, Vapnik V. Support-vector networks. Machine learning. 1995;20(3):273–297.

[95] Dredze M, Gevaryahu R, Elias-Bachrach A. Learning Fast Classifiers for Image Spam. In: CEAS; 2007. .

[96] Uemura M, Tabata T. Design and evaluation of a Bayesian-filter-based image spam filtering method. In: Information Security and Assurance, 2008. ISA 2008. International Conference on. IEEE; 2008. p. 46–51.

[97] Liu TJ, Tsao WL, Lee CL. A High Performance Image-Spam Filtering System. IEEE; 2010. p. 445–449. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5571607.

[98] Mandal MK, Idris F, Panchanathan S. A critical evaluation of image and video indexing techniques in the compressed domain. Image and Vision Computing. 1999;17(7):513–529. Available from: http://www.sciencedirect.com/science/article/pii/S0262885698001437.

[99] Lay JA, Guan L. Image retrieval based on energy histograms of the low frequency DCT coefficients. In: Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on. vol. 6. IEEE; 1999. p. 3009–3012. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=757474.

[100] Schaefer G. JPEG image retrieval by simple operators. 2nd International Workshop on Content Based Multimedia and Indexing. 2001;p. 207–214.

[101] Jiang J, Armstrong A, Feng GC. Direct content access and extraction from JPEG compressed images. Pattern Recognition. 2002;35(11):2511–2519. Available from: http://www.sciencedirect.com/science/article/pii/S0031320301002175.

[102] Lu ZM, Li SZ, Burkhardt H. A content-based image retrieval scheme in JPEG compressed domain. International Journal of Innovative Computing, Information and Control. 2006;2(4):831–839. Available from: http://lmb.informatik.uni-freiburg.de/papers/download/lu_li_bu_2006IJICIC.pdf.

[103] Eom M, Choe Y. Fast extraction of edge histogram in dct domain based on mpeg7. In: Proceedings of world academy of science, engineering and technology. vol. 9. Citeseer; 2005. p. 209–212. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.301.5040&rep=rep1&type=pdf.

[104] Shneier M, Abdel-Mottaleb M. Exploiting the JPEG compression scheme for image retrieval. IEEE Transactions on pattern Analysis and machine Intelligence. 1996;18(8):849–853. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=531805.

[105] Schaefer G, Edmundson D. DC Stream Based JPEG Compressed Domain Image Retrieval. In: Huang R, Ghorbani AA, Pasi G, Yamaguchi T, Yen NY, Jin B, editors. Active Media Technology. No. 7669 in Lecture Notes in Computer Science. Springer Berlin Heidelberg; 2012. p. 318–327. Available from: http://link.springer.com/chapter/10.1007/978-3-642-35236-2_32.

[106] Watson AB. Perceptual optimization of DCT color quantization matrices. In: Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference. vol. 1. IEEE; 1994. p. 100–104.

[107] Edmundson D, Schaefer G. Exploiting JPEG Compression for Image Retrieval. IEEE; 2012. p. 485–486. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6424712.

[108] Schaefer G, Edmundson D, Sakurai Y. Fast JPEG Image Retrieval Based on AC Huffman Tables. IEEE; 2013. p. 26–30. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6727165.

[109] Roussev V, Ahmed I, Barreto A, McCulley S, Shanmughan V. Cloud forensics–Tool development studies & future outlook. Digital Investigation. 2016 Sep;18:79–95. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287616300536.

[110] Taute B, Grobler M, Nare S. Forensic challenges for handling incidents and crime in cyberspace; 2009. Available from: https://researchspace.csir.co.za/dspace/bitstream/handle/10204/3756/Taute_d1_2009.pdf?sequence=1.

[111] Poisel R, Tjoa S. A Comprehensive Literature Review of File Carving. In: 2013 Eighth International Conference on Availability, Reliability and Security (ARES); 2013. p. 475–484.

[112] Larbanet A, Lerebours J, David JP. Detecting very large sets of referenced files at 40/100 GbE, especially MP4 files. Digital Investigation. 2015 Aug;14:S85–S94. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287615000560.

[113] Huiskes MJ, Thomee B, Lew MS. New trends and ideas in visual concept detection: the MIR flickr retrieval evaluation initiative. In: Proceedings of the international conference on Multimedia information retrieval. ACM; 2010. p. 527–536. Available from: http://dl.acm.org/citation.cfm?id=1743475.

[114] Garfinkel S, Farrell P, Roussev V, Dinolt G. Bringing science to digital forensics with standardized forensic corpora. Digital Investigation. 2009 Sep;6:S2–S11. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287609000346.

[115] jpegtran features;. Available from: http://jpegclub.org/jpegtran/.

[116] Clark A. Python-Pillow; 2015. Available from: https://github.com/python-pillow/Pillow.

[117] Independent JPEG Group. Libjpeg; 2016. Available from: http://www.ijg.org/.

[118] Portable Network Graphics (PNG) Specification (Second Edition) ISO/IEC 15948:2003; 2003. Available from: https://www.w3.org/TR/PNG/.

[119] w3techs. Usage Statistics of Image File Formats for Websites, July 2018; 2018. Available from: https://w3techs.com/technologies/overview/image_format/all.

[120] Deutsch P, Gailly JL. Zlib compressed data format specification version 3.3; 1996. RFC 1950.

[121] Jones D. PyPNG; 2016. Available from: https://github.com/drj11/pypng.

[122] Liu WJ. Empty Standby List; 2016. Available from: https://wj32.org/wp/software/empty-standby-list/.

[123] Skodras A, Christopoulos C, Ebrahimi T. The JPEG 2000 still image compression standard. Signal Processing Magazine, IEEE. 2001;18(5):36–58. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=952804.

[124] Mozilla. Mozjpeg: Improved JPEG encoder.; 2017. Available from: https://github.com/mozilla/mozjpeg.

[125] Alakuijala J, Obryk R, Stoliarchuk O, Szabadka Z, Vandevenne L, Wassenberg J. Guetzli: Perceptually Guided JPEG Encoder. arXiv preprint arXiv:170304421. 2017;Available from: https://arxiv.org/abs/1703.04421.

[126] Cryer J. Resemble.js: Image analysis and comparison. Huddle; 2017. Original-date: 2013-02-21T14:25:27Z. Available from: https://github.com/Huddle/Resemble.js.

[127] Ardizzone E, La Cascia M, Avanzato A, Bruna A. Video indexing using MPEG motion compensation vectors. In: Multimedia Computing and Systems, 1999. IEEE International Conference on. vol. 2. IEEE; 1999. p. 725–729.

[128] Lainema J, Hannuksela MM, Vadakital VKM, Aksu EB. HEVC still image coding and high efficiency image file format. In: 2016 IEEE International Conference on Image Processing (ICIP); 2016. p. 71–75.

[129] Tallis B. The ADATA XPG SX8200 & GAMMIX S11 NVMe SSD Review: High Performance At All Sizes; 2018. Available from: https://www.anandtech.com/show/13112/the-adata-sx8200-gammix-s11-nvme-ssd-review.

[130] Sealey P. Remote forensics. Digital Investigation. 2004 Dec;1(4):261–265. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287604000854.

[131] Seagate. Transition to Advanced Format 4K Sector Hard Drives | Seagate UK;. Available from: https://www.seagate.com/gb/en/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/.

[132] University of Southern California. Mediaq Videotake Dataset;. Available from: http://mediaq.usc.edu:8080/dataset/.

[133] Saarinen J. FBI Ordered to Copy 150 Terabytes of Data Seized From Megaupload; 2012. Available from: https://www.wired.com/2012/06/megaupoad-data/.

[134] Scanlon M, Kechadi MT. Online Acquisition of Digital Forensic Evidence. In: Goel S, editor. Digital Forensics and Cyber Crime. vol. 31. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 122–131. Available from: http://link.springer.com/10.1007/978-3-642-11534-9_12.

[135] Koopmans MB, James JI. Automated network triage. Digital Investigation. 2013 Sep;10(2):129–137. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287613000273.

[136] Homem I, Dosis S, Popov O. LEIA: The Live Evidence Information Aggregator: Towards efficient cyber-law enforcement. In: Internet Security (WorldCIS), 2013 World Congress on. IEEE; 2013. p. 156–161.

[137] Roussev V, Barreto A, Ahmed I. API-Based Forensic Acquisition of Cloud Drives. In: Advances in Digital Forensics XII. IFIP Advances in Information and Communication Technology. Springer, Cham; 2016. p. 213–235. Available from: http://link.springer.com/chapter/10.1007/978-3-319-46279-0_11.

[138] Yonan J. OpenVPN - Open Source VPN; 2002. Available from: https://openvpn.net/.

[139] Schweizer B. iops: Benchmark disk IOs; 2010. Original-date: 2011-09-03T13:08:48Z. Available from: https://github.com/cxcv/iops.

[140] Scanlon M. Battling the digital forensic backlog through data deduplication. In: Innovative Computing Technology (INTECH), 2016 Sixth International Conference on. IEEE; 2016. p. 10–14.

[141] Quick D, Tassone C, Choo KKR. Forensic Analysis of Windows Thumbcache files. Quick D, Tassone C and Choo KK R. 2014;Available from: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2429795.

[142] Parsonage H. Under My Thumbs – Revisiting Window s thumbnail databases and some new revelations about the forensic implications.; 2012. Available from: http://computerforensics.parsonage.co.uk/downloads/UnderMyThumbs.pdf.

[143] Morris S, Chivers H. A comparative study of the structure and behaviour of the operating system thumbnail caches used in Kubuntu and Ubuntu (9.10 and 10.04). Proceedings from 4th Cybercrime Forensics Education & Training Canterbury Christ Church University, Canterbury, UK. 2010;.

[144] Leom MD, DOrazio CJ, Deegan G, Choo KKR. Forensic Collection and Analysis of Thumbnails in Android. IEEE; 2015. p. 1059–1066. Available from: http://ieeexplore.ieee.org/document/7345391/.

[145] Newcomer S, Martin L. Determining User Actions In Os X Based On Quicklook Thumbnail Cache Database Entries. Issues in Information Systems. 2014;15(2).

[146] Kutcher E. Thumbcache Viewer - Extract thumbnail images from the thumbcache_*.db and iconcache_*.db database files.; 2016. Available from: https://thumbcacheviewer.github.io/.

[147] Brinkmann M. How to block the automatic cleaning of Windows 10's Thumbnail Cache - gHacks Tech News; 2019. Available from: https://www.ghacks.net/2019/03/04/how-to-block-the-automatic-cleaning-of-windows-10s-thumbnail-cache/.

[148] Khatri Y. Windows 7 Thumbcache hash algorithm; 2012. Available from: http://www.swiftforensics.com/2012/06/windows-7-thumbcache-hash-algorithm.html.

[149] McKeown. Shellthumbs; 2018. Available from: https://github.com/smck1/shellthumbs.

[150] Dandass YS, Necaise NJ, Thomas SR. An Empirical Analysis of Disk Sector Hashes for Data Carving. J Digit Forensic Pract. 2008 Apr;2(2):95–104. Available from: http://dx.doi.org/10.1080/15567280802050436.

[151] Kuksov I. What EXIF can tell about the photos you post online; 2016. Available from: https://www.kaspersky.co.uk/blog/exif-privacy/7893/.

[152] Klinger E, Starkweather D. pHash – the open source perceptual hash library; 2012. Available from: http://www.phash.org/apps/.

[153] Buchner J. ImageHash; 2017. Available from: https://pypi.org/project/ImageHash/.

[154] Yang B, Gu F, Niu X. Block mean value based image perceptual hashing. In: Intelligent Information Hiding and Multimedia Signal Processing, 2006. IIH-MSP'06. International Conference on. IEEE; 2006. p. 167–172. Available from: http://ieeexplore.ieee.org/abstract/document/4041692/.

[155] Commonsmachinery. Contribute to blockhash development by creating an account on Github. Commons Machinery; 2018. Original-date: 2014-09-02T17:46:34Z. Available from: https://github.com/commonsmachinery/blockhash.

[156] James JI, Gladyshev P. A survey of digital forensic investigator decision processes and measurement of decisions based on enhanced preview. Digital Investigation. 2013 Sep;10(2):148–157. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1742287613000340.

[157] Grispos G, Glisson WB, Storer T. Using smartphones as a proxy for forensic evidence contained in cloud storage services. In: System Sciences (HICSS), 2013 46th Hawaii International Conference on. IEEE; 2013. p. 4910–4919.

[158] Dropbox Inc . GitHub - dropbox/dropbox-sdk-python: Python SDK for Dropbox API v2.;. Available from: https://github.com/dropbox/dropbox-sdk-python.

[159] Requests: HTTP for Humans — Requests 2.19.1 documentation;. Available from: http://docs.python-requests.org/en/master/.

[160] Free up drive space in Windows 10 - Windows Help;. Available from: https://support.microsoft.com/en-us/help/12425/windows-10-free-up-drive-space.

[161] Morris S, Chivers H. Forming a Relationship between Artefacts identified in thumbnail caches and the remaining data on a storage device. Cybercrime Forensics Education and Training. 2011;Available from: https://www.researchgate.net/profile/Sarah_Morris7/publication/262327215_Forming_a_Relationship_between_Artefacts_identified_in_thumbnail_caches_and_the_remaining_data_on_a_storage_device/links/00b495374dfe6b0f09000000.pdf.

[162] Gillam WB, Rogers M. File Hound: A Forensics Tool for First Responders. In: DFRWS; 2005. p. 1–7.

[163] Dalins J, Wilson C, Carman M. Monte-Carlo Filesystem Search – A crawl strategy for digital forensics. Digital Investigation. 2015 Jun;13:58–71. Available from: https://linkinghub.elsevier.com/retrieve/pii/S1742287615000420.

[164] Wu AQ, Kubota Y, Klemmer T, Rausch T, Peng C, Peng Y, et al. HAMR Areal Density Demonstration of 1+ Tbpsi on Spinstand. IEEE TRANSACTIONS ON MAGNETICS. 2013;49(2):779.

[165] Re M. HAMR: the Next Leap Forward is Now; 2017. Available from: https://blog.seagate.com/craftsman-ship/hamr-next-leap-forward-now/.

[166] Feist J. Multi Actuator Technology: A New Performance Breakthrough; 2017. Available from: https://blog.seagate.com/craftsman-ship/multi-actuator-technology-a-new-performance-breakthrough/.

[167] McKeown S. Bulk-Bing-Image-downloader: Download full sized images returned from bing image search; 2016. Original-date: 2016-04-07T12:45:36Z. Available from: https://github.com/smck1/Bulk-Bing-Image-downloader.

[168] Miyazaki N. CrystalDiskMark; 2017. Available from: https://crystalmark.info/software/CrystalDiskMark/manual-en/.

Appendix A

Dataset Creation

A.1 Chapter 3 Datasets

A.1.1 Govdocs PNG Conversion

The following script was used to convert the original Govdocs JPEG files to the PNG format:

```
# Python 2
import os
from sys import argv
from PIL import Image, ImageEnhance
from copy import copy
from thread_utils import *

def saveAsPNG(image, fname, outpath):
    # Convert the source file to a PNG with no other modification
    fpath = os.path.join(outpath, fname + '.png')
    try:
        image.save(fpath)
    except IOError, m:
        print "Saving as PNG failed for: {}. \nReason:{}".format(fname
    , m)


def generatemods(filepath, outpath):
    # Generate and save image modfiications for a source file, save
   them to outpath.
    originalimage = Image.open(filepath)
    fname, ext = os.path.splitext(filepath)
    fname = fname.split(os.sep)[-1]
    saveAsPNG(originalimage,fname, outpath)
```

```


# Parse commandline args.
path = ''

if len(argv) != 3:
    print "usage: <original_images_dir> <output_dir>"
    exit()
elif len(argv) == 3:
    path = argv[1]
    outpath = argv[2]


if not os.path.exists(outpath):
    print "Output directory does not exist: Creating."
    os.makedirs(outpath)

# Modify all items in the path, save mods to outpath
tpool = ThreadPool(4)
count = 0
print "Beginning conversion..."
for subdir, dirs, files in os.walk(path):
    for f in files:
        try:
            fpath = os.path.join(subdir, f)
            tpool.add_task(generatemods, fpath, outpath)
            count +=1
            if count % 1000 ==0:
                print "Converted:", count
        except Exception:
            print "problem processing: {}".format(f)

tpool.wait_completion()
```

Listing A.1 Code used to generate PNGs from the Govdocs dataset. Does not include de-duplication code.

A.1.2 Difference From Original Govdocs

During the conversion four files failed to convert using the Python PIL library, and one PNG had an invalid signature post conversion. These images, as well as the 341 binary duplicates, as determined by SHA256, were removed from the modified Govdocs PNG corpus.

4 Failed to Convert:

422364.jpg 656998.jpg 657390.jpg 657911.jpg

1 Invalid Signatures After Conversion:

800992.png

341 Duplicates Removed:

002640.png 007377.png 010046.png 013735.png 015118.png 015277.png 020741.png 023261.png
025447.png 025758.png 026212.png 027143.png 028261.png 028781.png 028788.png 033480.png
033491.png 033501.png 033878.png 035425.png 035428.png 035429.png 035698.png 035711.png
035963.png 040830.png 051322.png 051575.png 054012.png 059280.png 075350.png 075525.png
076436.png 081112.png 081120.png 088427.png 089439.png 092515.png 092521.png 092554.png
092560.png 092853.png 097013.png 097015.png 099447.png 102379.png 109760.png 110742.png
112399.png 112424.png 116271.png 118532.png 123259.png 126780.png 128269.png 128270.png
128271.png 134314.png 141138.png 141140.png 143202.png 148871.png 151078.png 153343.png
157541.png 158783.png 169422.png 169431.png 169446.png 171921.png 175167.png 183029.png
186142.png 186145.png 191994.png 193151.png 193925.png 194473.png 200586.png 212149.png
227152.png 229394.png 231767.png 232902.png 233980.png 233995.png 234400.png 238224.png
242215.png 242220.png 243429.png 243496.png 244139.png 245334.png 249151.png 258948.png
258954.png 263535.png 268507.png 268522.png 268525.png 268526.png 268535.png 268536.png
268544.png 270646.png 276140.png 278118.png 281157.png 281171.png 281452.png 289382.png
290530.png 292970.png 294564.png 295387.png 298758.png 301730.png 307942.png 309251.png
311477.png 311481.png 312364.png 317063.png 320369.png 320393.png 320400.png 320404.png
320406.png 320469.png 320476.png 323509.png 330080.png 800987.png 336340.png 336457.png
336877.png 339173.png 340375.png 340869.png 342450.png 346546.png 346556.png 350026.png
350033.png 350912.png 352542.png 354082.png 356369.png 357481.png 359126.png 360131.png
362044.png 362171.png 365914.png 366063.png 369034.png 369069.png 372481.png 391374.png
397626.png 406967.png 409013.png 411553.png 416428.png 419991.png 419996.png 420002.png
424848.png 424849.png 426030.png 427438.png 431807.png 434513.png 451838.png 452550.png
461857.png 496974.png 497019.png 504002.png 517692.png 539724.png 545202.png 546945.png
593225.png 656806.png 657195.png 657383.png 657579.png 657748.png 658091.png 658093.png
658261.png 658422.png 658586.png 660548.png 661092.png 661434.png 662178.png 662542.png
663262.png 663450.png 663594.png 663774.png 663941.png 663956.png 663962.png 664118.png
664297.png 664468.png 666005.png 666008.png 666025.png 666184.png 666512.png 666832.png
666837.png 666842.png 666995.png 666996.png 667001.png 667026.png 667775.png 667787.png
667944.png 668251.png 668549.png 668556.png 668563.png 668570.png 668578.png 668579.png

669170.png 669171.png 669179.png 669181.png 669192.png 669193.png 669475.png 670250.png 670258.png 670411.png 670560.png 670563.png 680636.png 684187.png 689886.png 699323.png 707828.png 711544.png 727522.png 730448.png 731138.png 731145.png 731165.png 732812.png 733836.png 734130.png 734133.png 735907.png 736494.png 739601.png 741018.png 742614.png 744600.png 750446.png 753422.png 755258.png 756595.png 759249.png 765086.png 765130.png 766497.png 767968.png 768488.png 771076.png 771480.png 771880.png 771965.png 774339.png 776333.png 777969.png 781506.png 782306.png 782499.png 785468.png 785484.png 786607.png 786874.png 787095.png 788218.png 789220.png 791965.png 793603.png 798385.png 800828.png 805080.png 816130.png 817021.png 824163.png 827857.png 834317.png 837064.png 841849.png 842559.png 842652.png 848446.png 848940.png 849413.png 849918.png 852933.png 860905.png 861030.png 861484.png 863898.png 864842.png 869082.png 874503.png 879148.png 880347.png 880466.png 886981.png 887543.png 892418.png 894046.png 897532.png 899735.png 907491.png 908342.png 908659.png 926130.png 934431.png 938845.png 941856.png 942081.png 942509.png 951476.png 967540.png 968637.png 970091.png 989083.png

A.1.3 Bing Image Collection

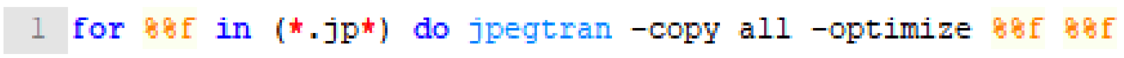
The Bing dataset was collected by issuing queries to to the Bing searching engine and saving the results, with binary level de-duplication. Downloading was handled by a fork of the Bing Image Downloader [167]. This works by obtaining query results from Bing, parsing URLs in the results, then downloading the images from the original source page. Results were filtered for the PNG extension prior to downloading. 107 unique queries across a variety of topics were issued, collecting images from over 2750 unique domains. Queries were intended to capture a wide variety of natural and synthetic images, and included terms relating to art, scenery, music, technology, games, wildlife, and television.

A.1.4 Govdocs Optimised

This dataset is a transformation of the original Govdocs dataset, with each image being modified to include optimised Huffman tables. Optimisation makes use of the `jpegtran` [115] utility, developed by the Independent JPEG Group (http://ijg.org/). All existing markers and metadata were copied (`-copy all`, only the Huffman tables were modified in the images. By default, this command outputs baseline JPEGs, converting progressive and extended JPEGs to this format.

5 Failed to optimise

658899.jpg 656998.jpg 657390.jpg 657911.jpg 658423.jpg



```
for %%f in (*.jp*) do jpegtran -copy all -optimize %%f %%f
```

Fig. A.1 Windows command line code to optimise the Huffman tables of all JPEGs in a given directory. Makes use of the jpegtran utility.

A.2 Chapter 4 Datasets

This chapter uses the Flickr 1 Million and Govdocs PNG datasets as they were in Chapter 3. Details here pertain to the 25,000 subsets used in the networked storage experiments.

A.2.1 Flickr Subset

The code snippet below was used to enumerate Flickr files in numerical order, which were subsequently copied to a different directory.

```python
# Python 2
flist = []
print "Generating input paths for {} files.".format(args.filecount)
for x in xrange(0, args.filecount):
    fname = "{}.jpg".format(x)
    flist.append(fname)
```

Listing A.2 Code used to subset the Flickr 1 Million dataset in numerical order.

A.2.2 Flickr Subset PNG

This dataset was created by creating a copy of the Flickr Subset above (Section A.2.1), before converting to PNG using the script in Section A.1.1.

A.2.3 Govdocs Subset PNG

This subset was created from the Govdocs PNG conversion in Section A.1.1. Govdocs file names are numerical, and fixed length. As such, the Python `os.listdir` function was sufficient to obtain a numerically ordered list, as they were contained in the original zip file. This list was then sliced to obtain the first 25,000 items, which were copied to a new directory.

A.3 Chapter 5 Datasets

Datasets in this chapter are derived from the Flickr 1 Million dataset. Images are converted to Windows specific thumbnails, as described in Section 5.4.1. The original Flickr

1 Million dataset is only used ‘as is’ during the comparisons of full sized images in Section 5.6.2.

A.4 A Note on File Size Distributions

The Govdocs dataset and its derivatives have file sizes which are clearly not normally distributed, with a long tail towards higher file sizes. However, for the sub-file experiments, the mean file size is what is important. This is because sub-file approaches are insensitive to fill size, while the overall amount of data to process is the primary limitation of the full file hashing approach. File access overheads remain the same, regardless of file size distribution. As such, while the distribution of file sizes is worth noting, it should not affect the results of the benchmarks in Chapters 3 and 4.

Appendix B

Benchmark Configurations

B.1 Local Disk Experiments

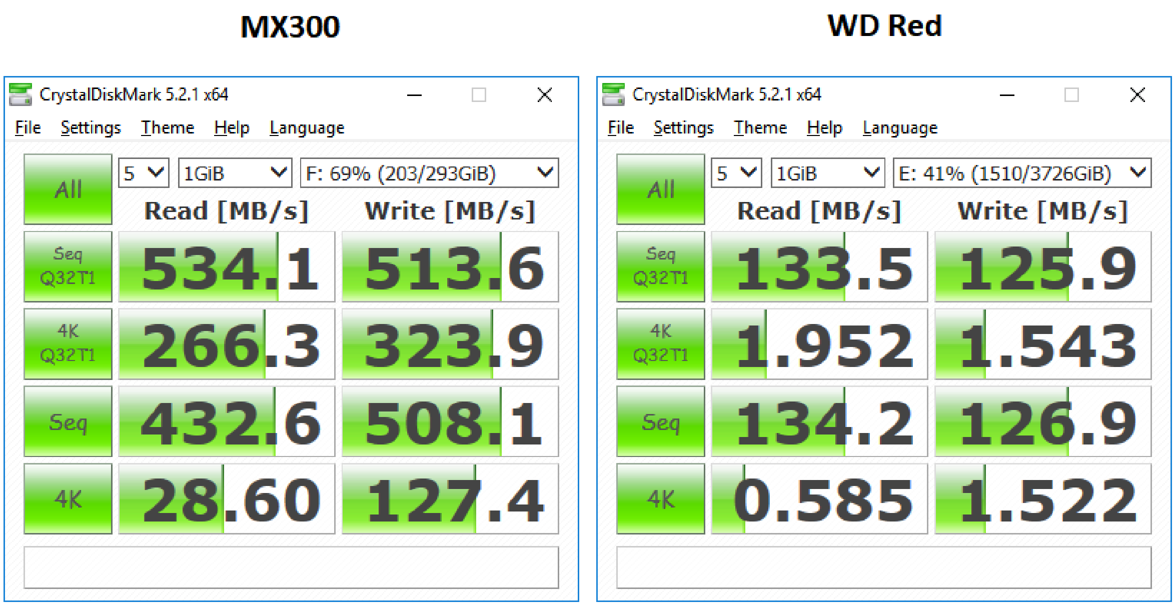
B.1.1 Machine Configuration

Machine	Specification	Software
Workstation	CPU: Intel Core i5-4690k (4 Core) RAM: 16GiB DDR3 RAM HDD: Western Digital Red 4TB SSD: Crucial MX300 525GB (OS Drive not used in experiments)	**Windows:** Windows 10 64 Bit Python 2.7.12 Visual Studio 15.0.26430.16 **Linux PNG/JPEG:** Ubuntu 15.04 LTS Python 2.7.9 g++ 4.8.4 **Linux Generic Sub-file:** Ubuntu 16.04 LTS Python 2.7.12
Laptop	CPU: Intel Core i7-5500U (2 Core) RAM: 8GiB DDR3 RAM HDD: N/A SSD: Samsung 840 EVO 500GB(OS)	Windows 10 64 Bit Python 2.7.12
Netbook	CPU: Intel Atom N450 (1 Core, 2 Threads) RAM: 1GiB DDR2 HDD: Western Digital 160GB(OS) SSD: N/A	Windows 10 32 Bit

Table B.1 Hardware and Software configuration of the computers used in the local disk experiments, for PNG, JPEG, Local generic sub-file and thumbnail. Both the laptop and netbook only have a single drive running the OS, while the workstation runs the OS on a separate drive not involved in the benchmark.

B.1.2 Storage Media Benchmarks

Disk benchmarks as provided by the CrystalDiskMark utility [168]. Note: Values are in MB/s rather than MiB/s, and have been converted for tables in Section 3.6.3.



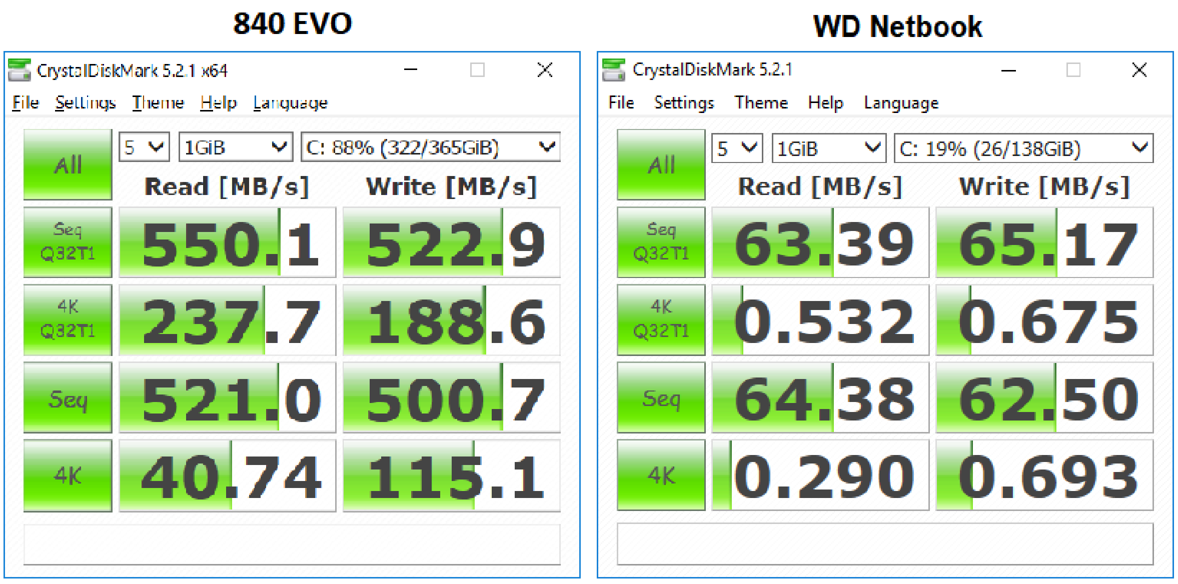
MX300

```
CrystalDiskMark 5.2.1 x64
File  Settings  Theme  Help  Language
All   5   1GiB   F: 69% (203/293GiB)
             Read [MB/s]   Write [MB/s]
Seq Q32T1    534.1         513.6
4K Q32T1     266.3         323.9
Seq          432.6         508.1
4K           28.60         127.4
```

WD Red

```
CrystalDiskMark 5.2.1 x64
File  Settings  Theme  Help  Language
All   5   1GiB   E: 41% (1510/3726GiB)
             Read [MB/s]   Write [MB/s]
Seq Q32T1    133.5         125.9
4K Q32T1     1.952         1.543
Seq          134.2         126.9
4K           0.585         1.522
```

Fig. B.1 Workstation storage benchmarks. SSD on the left, HDD on the right).



840 EVO

```
CrystalDiskMark 5.2.1 x64
File  Settings  Theme  Help  Language
All   5   1GiB   C: 88% (322/365GiB)
             Read [MB/s]   Write [MB/s]
Seq Q32T1    550.1         522.9
4K Q32T1     237.7         188.6
Seq          521.0         500.7
4K           40.74         115.1
```

WD Netbook

```
CrystalDiskMark 5.2.1
File  Settings  Theme  Help  Language
All   5   1GiB   C: 19% (26/138GiB)
             Read [MB/s]   Write [MB/s]
Seq Q32T1    63.39         65.17
4K Q32T1     0.532         0.675
Seq          64.38         62.50
4K           0.290         0.693
```

Fig. B.2 Laptop SSD (left) and Netbook HDD (right) storage benchmarks.

B.2 Network Drive Experiments

B.2.1 Machine Configuration

Machine	**Specification**	**Software**
Client Workstation	HP EliteDesk 800 G1 i5-4590s, 4GiB RAM	Windows 7 Enterprise 64bit NFS v3 client OpenVPN Client (AES 256/SHA256, no compression, 2048bit key)
LAN Server (Hypervisor)	2× Intel Xeon E5-2697v4 384 GiB RAM RAID 10 10× 840 EVO 1TB SSD	vSphere 6.5.0 ESXi 6.5.0
LAN Virtual Machine	ESXi 6.5+ Virtual Machine 1 Virtual CPU, 2GiB RAM 500GB SSD storage (EXT4)	Ubuntu 17.04 LTS Samba server 4.3.11-ubuntu NFS server 1.2.8-9.2ubuntu2
Internet Server	Digital Ocean $10 Droplet 1 Virtual CPU, 2GiB RAM 50GB SSD storage (EXT4)	Ubuntu 16.04 LTS Samba server 4.5.8-ubuntu NFS server 1:1.2.8-9ubuntu12.1

Table B.2 Specifications of the equipment and software set-up for the networked generic sub-file benchmarks. This table is a copy of Table 4.7.

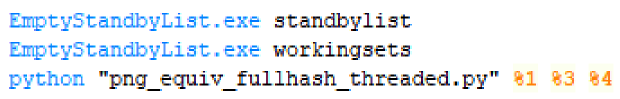
Appendix C

PNG Signature - Supplementary

C.1 Code Snippets

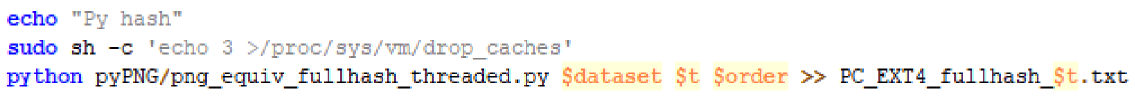
C.1.1 Cache Clearing and Repeat Experiments

It was important to clear the operating system memory cache between runs. Figure C.1 depicts a typical iteration of a benchmark on Windows. The EmptyStandbyList utility [122] was used to clear both the working set and standby list of Windows. On Linux, the page-cache, dentries and inodes were flushed by passing the appropriate flag to drop_caches, as depicted in Figure C.2.



```
EmptyStandbyList.exe standbylist
EmptyStandbyList.exe workingsets
python "png_equiv_fullhash_threaded.py" %1 %3 %4
```

Fig. C.1 A typical iteration of the benchmark on Windows, with memory cache flushing.



```
echo "Py hash"
sudo sh -c 'echo 3 >/proc/sys/vm/drop_caches'
python pyPNG/png_equiv_fullhash_threaded.py $dataset $t $order >> PC_EXT4_fullhash_$t.txt
```

Fig. C.2 A typical iteration of the benchmark on Ubuntu, with memory cache flushing

C.1.2 PyPNG Sub-file Signature Generation and Benchmark

The Python code below contains the code to create thread pools, handle file ordering, and time the execution to extract sub-file PyPNG signatures. thread_utils is simply a helper for the creation of thread pools and thread safe objects.

```
# Python 2
import png
import os
from sys import argv
import hashlib
import numpy as np
import time
import random
from thread_utils import *


# IHDR feature subset
features = ['height', 'bitdepth', 'compression',
 'width', 'interlace',
 'color_type', 'signature','filter']


def processFile(dirpath, impath, size, stringstore):
    fpath = os.path.join(dirpath, impath)
    try:
        fil = open(fpath, 'rb')
        r = png.Reader(file=fil)
        r.preamble() # this gets all of the header features
        d = r.file.read(size)
        fil.close()
        idatlength = r.atchunk[0] # length of first IDAT
        info = ''
        for f in features:  # subset features to IHDR only
            info+=str(r.__dict__.get(f, ''))
        info+=str(idatlength)
        info+=d

        with stringstore as i:
            if i.has_key(info):
                i[info].append(impath)
            else:
                i[info] = [impath]
        return 1

    except Exception, e:
        print fpath, e

# Parse commandline args.
path = ''
bytelength = ''
possible_orders=['normal', 'random', 'reverse']
```

```
order = possible_orders[0]
size = 0
num_threads=1
if len(argv) == 4:
    path = argv[1]
    num_threads = int(argv[3])
    size = int(argv[2])
elif len(argv) == 5:
    path = argv[1]
    size = int(argv[2])
    num_threads =  int(argv[3])
    order = argv[4]
    if order not in possible_orders:
        print "invalid order: {}".format(order)
        exit()
else:
    print "args: <file_directory> <idat data size>
    <num_threads> <(optional) ordering>"
    exit()


infostrings = ThreadSafeDict()
tpool = ThreadPool(num_threads)
classes = {}
toosmall = 0
toosmallandnotunique = 0
psusage = []
counter= 0




flist = os.listdir(path)
if order == 'reverse':
    flist.reverse()
elif order == 'random':
    random.shuffle(flist)

t0 = time.time()
for im in flist:
    tpool.add_task(processFile, path, im, size, infostrings)
tpool.wait_completion()

# timer end
t1 = time.time()
total_time = t1-t0
```

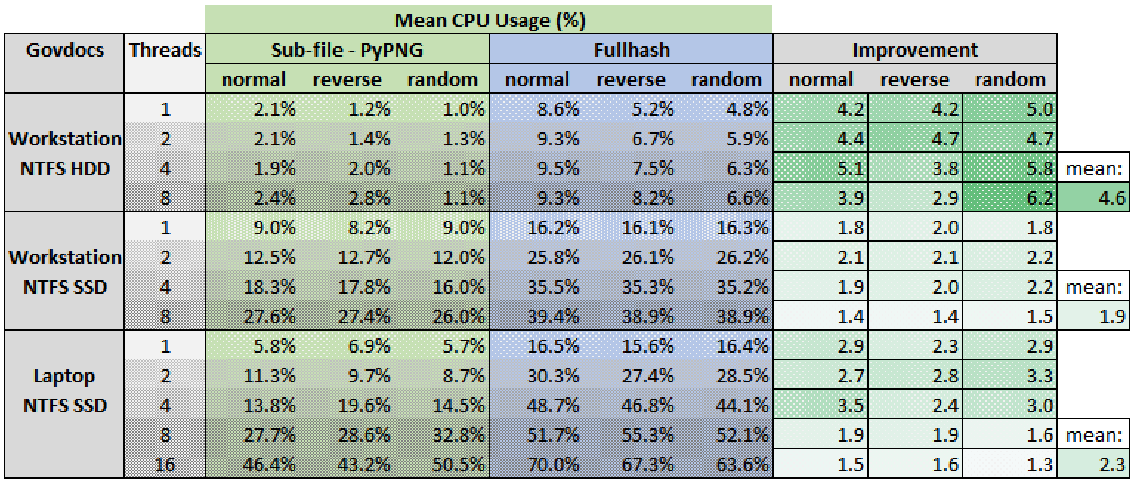
```

siglengths = []
for s in infostrings:
    siglengths.append(len(s))

    classes[len(infostrings[s])]= classes.get(len(infostrings[s]),0)
   +1

sl = siglengths


print "IHDR_IDAT–Length_{}B_{}  Total time: {}".
format(size,order, total_time)
```

Listing C.1 Benchmark for PyPNG sub-file signature.

C.2 Supplementary Results

C.2.1 CPU Usage Data

CPU usage data was collected for the PNG experiments, depicted below in Figure C.3. Very little overall CPU utilisation was seen on the hard drive, while the lower specification laptop processor began to reach capacity with high thread counts on fullhash.



		Mean CPU Usage (%)									
Govdocs	Threads	Sub-file - PyPNG			Fullhash			Improvement			
		normal	reverse	random	normal	reverse	random	normal	reverse	random	
Workstation NTFS HDD	1	2.1%	1.2%	1.0%	8.6%	5.2%	4.8%	4.2	4.2	5.0	
	2	2.1%	1.4%	1.3%	9.3%	6.7%	5.9%	4.4	4.7	4.7	
	4	1.9%	2.0%	1.1%	9.5%	7.5%	6.3%	5.1	3.8	5.8	mean:
	8	2.4%	2.8%	1.1%	9.3%	8.2%	6.6%	3.9	2.9	6.2	4.6
Workstation NTFS SSD	1	9.0%	8.2%	9.0%	16.2%	16.1%	16.3%	1.8	2.0	1.8	
	2	12.5%	12.7%	12.0%	25.8%	26.1%	26.2%	2.1	2.1	2.2	
	4	18.3%	17.8%	16.0%	35.5%	35.3%	35.2%	1.9	2.0	2.2	mean:
	8	27.6%	27.4%	26.0%	39.4%	38.9%	38.9%	1.4	1.4	1.5	1.9
Laptop NTFS SSD	1	5.8%	6.9%	5.7%	16.5%	15.6%	16.4%	2.9	2.3	2.9	
	2	11.3%	9.7%	8.7%	30.3%	27.4%	28.5%	2.7	2.8	3.3	
	4	13.8%	19.6%	14.5%	48.7%	46.8%	44.1%	3.5	2.4	3.0	
	8	27.7%	28.6%	32.8%	51.7%	55.3%	52.1%	1.9	1.9	1.6	mean:
	16	46.4%	43.2%	50.5%	70.0%	67.3%	63.6%	1.5	1.6	1.3	2.3

Fig. C.3 CPU Usage data for Fullhash and the sub-file PyPNG approach. Largest benefits seen on HDD at lower thread counts. The sub-file approach uses less CPU than full file hashing in all cases.

Appendix D

JPEG Signature - Supplementary

D.1 JPEG Run-length Pattern

The JPEG standard makes use of a serpentine pattern (Figure D.1) when run-length encoding the 63 AC coefficients of the DCT matrix. Items in the bottom right are heavily quantized, and should often be zeroes. The serpentine pattern attempts to capitalise on this by producing many consecutive zeroes for efficient run-length encoding.

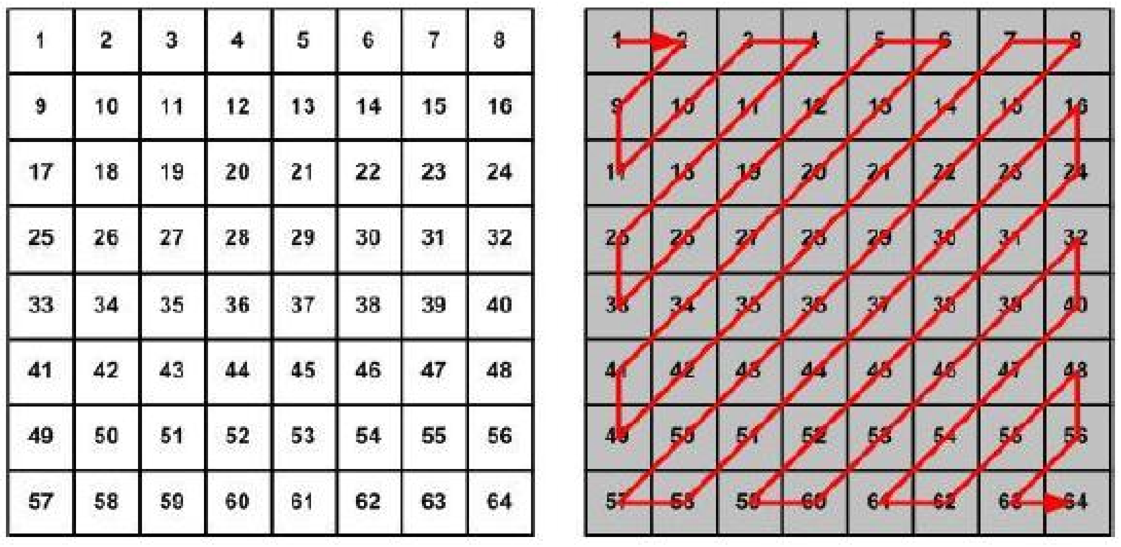


Fig. D.1 The zig-zag pattern used to run length encode DCT matrix coefficients in JPEG compression. Sourced from: https://commons.wikimedia.org/wiki/File:Zigzag_scanning.jpg

D.2 Code Snippets

D.2.1 Linux Volume Mounting Options

Volumes were mounted -ro,noatime in Ubuntu, however this comes with some implicit, version specific, options. The following output was obtained by running the `mount` command with no parameters after setting up the experiment.

NTFS: `ro,noatime,user_id=0,group_id=0,allow_other,blksize=4096`
EXT4: `ro,noatime,norecovery`

D.2.2 Huffman Sub-file JPEG Signature Generation and Benchmark

The following contains the C++ code for extracting Huffman signatures and performing the timed benchmark. The main body makes use of the Boost library for thread pools, and the main loop has a limit of 30000 items in the processing queue to avoid crashes. Code to read the JPEG is a modified version of the libjpeg example (https://github.com/LuaDist/libjpeg/blob/master/example.c), while the remaining code constructs a signature string by processing the Huffman length and value arrays.

```cpp
#include <boost/asio/io_service.hpp>
#include <boost/bind.hpp>
#include <boost/thread/thread.hpp>
#include <stdio.h>
#include <jpeglib.h>
#include <setjmp.h>
#include <iostream>
#include <vector>
#include <fstream>
#include <unordered_map>
#include <string>
#include <chrono>
#include <thread>

using namespace std;

// Global counters
atomic<int> completeCounter{0};

void appendArray(UINT8 arr[], int num_elements, string &s) {
    for ( int i = 0; i < num_elements; i++ ) {
      s.append(to_string(arr[i]));
    }
}
```

```

// For Length array in Huffman table. Append the values to s and
    return number of codes.
int appendLengths(UINT8 arr[], string & s) {
    // Always 17 elements, arr[0] should always be zero, skip it.
    int codecount = 0;
    for ( int i = 1; i < 17; i++ ) {
      s.append(to_string(arr[i]));
      codecount = codecount + arr[i];
    }
    return codecount;
}

// Print the huffman struct using printArray, return total number of
    codes
int appendHuffStruct(JHUFF_TBL table, string & s){
  int total_codes;
  // Get total number of codes and append the length array
  total_codes = appendLengths(table.bits, s);
  // Append Huffman values in length order.
  appendArray(table.huffval, total_codes, s);
  return total_codes;
}


string buildHuffString(jpeg_decompress_struct * cinfo){
  // Get the number of AC and DC huffman tables from the component
    information.
  // Note: this is only to the first SOS marker, there may be multiple
     in progressive JPEG
  int numDC = 0;
  int numAC = 0;
  for (int i=0; i < cinfo->num_components; i++){
    int current_ac;
    int current_dc;
    // The highest index+1 = number AC/DC tables in use.
    current_dc = cinfo->comp_info[i].dc_tbl_no;
    if (current_dc+1 > numDC){
      numDC = current_dc +1;
    }
    current_ac = cinfo->comp_info[i].ac_tbl_no;
    if (current_ac+1 > numAC){
      numAC = current_ac +1;
    }
  }

  string huffstring;
  /* Reserve memory to avoid loads of copies and moves
  There are only 348 possible codes, but they vary in length if int.
  */
  huffstring.reserve(2000);
  int total_codes = 0;

  // Append DC
  //vector<int> lenDCs;
  for (int i =0; i < numDC; i++){
    int l = appendHuffStruct(*cinfo->dc_huff_tbl_ptrs[i], huffstring);
    total_codes += l;
  }

  // Append AC
  for (int i =0; i < numAC; i++){
    int l = appendHuffStruct(*cinfo->ac_huff_tbl_ptrs[i], huffstring);
    total_codes +=l;
  }

  return huffstring;
} //end buildhuff




struct my_error_mgr {
  struct jpeg_error_mgr pub;/* "public" fields */

  jmp_buf setjmp_buffer;/* for return to caller */
};

typedef struct my_error_mgr * my_error_ptr;

/*
 * Here's the routine that will replace the standard error_exit method
   :
 */

METHODDEF(void)
my_error_exit (j_common_ptr cinfo)
{
  /* cinfo->err really points to a my_error_mgr struct, so coerce
    pointer */
  my_error_ptr myerr = (my_error_ptr) cinfo->err;

  /* Always display the message. */
  /* We could postpone this until after returning, if we chose. */
  //(*cinfo->err->output_message) (cinfo);
  /* Return control to the setjmp point */
  longjmp(myerr->setjmp_buffer, 1);
}


GLOBAL(string)
read_JPEG_file (char * filename)
{
  int status; // Status flag for header read

  /* This struct contains the JPEG decompression parameters and
    pointers to
   * working space (which is allocated as needed by the JPEG library).
   */
  struct jpeg_decompress_struct cinfo;
  struct my_error_mgr jerr;
  FILE * infile;/* source file */

  if ((infile = fopen(filename, "rb")) == NULL) {
    fprintf(stderr, "can't open %s\n", filename);
    string retstring = "failed";
    return retstring;
  }

  /* Step 1: allocate and initialize JPEG decompression object */
  cinfo.err = jpeg_std_error(&jerr.pub);
  jerr.pub.error_exit = my_error_exit;
  /* Establish the setjmp return context for my_error_exit to use. */
  if (setjmp(jerr.setjmp_buffer)) {
    /* If we get here, the JPEG code has signaled an error.
     * We need to clean up the JPEG object, close the input file, and
    return.
     */
    jpeg_destroy_decompress(&cinfo);
    fclose(infile);
    cerr << "Error: Failed to process " << filename <<endl;
    string retstring = "failed";
    return retstring;
  }

  /* Now we can initialize the JPEG decompression object. */
  jpeg_create_decompress(&cinfo);
  jpeg_stdio_src(&cinfo, infile);
  status = jpeg_read_header(&cinfo, TRUE); // ignore return code with
    void.

  fclose(infile);


  // Build Huffman String and destroy compression object.
  string hstring = buildHuffString(&cinfo);
  jpeg_destroy_decompress(&cinfo);
  return hstring;
}

int extractHuff(string filename){
      string sig;
      sig = read_JPEG_file(filename.c_str());
      if (sig == "failed"){
        // file failed to process
        cout << "failed " << filename << endl;
        completeCounter++;
        return 0;
      }
      completeCounter++;
      return 0;
}


/// Main ////////////////////////////
/////////////////////////////////////
int main(int argc, char *argv[]){
  if (argc != 3){
    cout << "Usage: <file_list_path, num_threads>" << endl;
    return 1;
  }
  cout << endl;
  // Assign thread count.
  int my_thread_count = stoi(argv[2]);


  vector<string> filenames;
  ifstream input( argv[1] );
  if (!input.is_open()){
    cout << "Couldn't open " << argv[1] << endl;
    return 1;
  }
  for( string line; getline( input, line ); )
  {
    //for each line, add it to the list of files
    filenames.push_back(line);
  }

  // Start timing
  time_t start = time(0);


  /*
   * Create an asio::io_service and a thread_group (through pool in
   essence)
   */
  boost::asio::io_service ioService;
  boost::thread_group threadpool;
  boost::asio::io_service::work work(ioService);

  /*
   * Create worker threads
   */
  for (int i = 0; i < my_thread_count; ++i)
    threadpool.create_thread(boost::bind(&boost::asio::io_service::run
   , &ioService));

  int c=0;
  for (size_t i=0; i<filenames.size(); i++){
  // Stop crashes from the pool filling up too much, but keep the
   threads fed.
    if (c<30000){
        ioService.post(boost::bind(extractHuff, filenames[i]));
        c++;
    } else {
        c = i − completeCounter;
        if (c >30000){
            std::this_thread::sleep_for(std::chrono::milliseconds(50))
   ;
        }
        ioService.post(boost::bind(extractHuff, filenames[i]));
    }
  }

  ioService.reset();
  while (completeCounter < filenames.size()){
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }
  ioService.stop();

  // End timer
  time_t end = time(0);
  double time = difftime(end, start);
  cout << "Processed: " << +completeCounter << endl;
  cout << "Time taken: " << time <<"s" << endl;

    return 0;
}
```

Listing D.1 Benchmark Huffman sub-file signature.

Appendix E

Generic Sub-file - Supplementary

Supplementary materials for Chapter 4.

E.1 Code Snippets

E.1.1 Linux Local Mounting Options

Volumes were mounted -ro,noatime in Ubuntu, however this comes with some implicit, version specific, options. The following output was obtained by running the `mount` command with no parameters after setting up the experiment.

NTFS: `ro,noatime,user_id=0,group_id=0,allow_other,blksize=4096`
EXT4: `ro,noatime,data=ordered`

E.1.2 Benchmark Code

The following Python code contains all of the necessary elements to run the sub-file benchmarks for both local and remote storage. The main loop and thread pools are essentially unchanged from the PNG code in Appendix C. This code calls the hash functions depicted in Section E.1.3, below.

```python
# Python 2
import os
from sys import argv
import hashlib
import time
from thread_utils import *
import argparse
from hashfuns import hashFull, hashFirstn, hashLastn, hashFirstLastn

```

```python
BLOCKSIZE=4096

# Set up args
parser = argparse.ArgumentParser(description='Run a benchmark of sub-file (or full file) hashing using SHA256.')
parser.add_argument('dir', help="Directory of files to process.")
parser.add_argument('--threads', '-t', type=int, nargs='?', const=1, help='number of threads to use. Default 1.')
parser.add_argument('--fullhash', '-full', action='store_true', help='perform full file hashing')
parser.add_argument('--firstn', '-fn', type=int, help="hash the FIRST n blocks, provide n.")
parser.add_argument('--lastn', '-ln', type=int, help="hash the LAST n blocks, provide n.")
parser.add_argument('--firstlastn', '-fl', type=int, help='hash the FIRST AND LAST n blocks, provide n.')
parser.add_argument('--subset', '-s', type=int, help='subset the files in the diectory, only process the first n files, provide n.')

args = parser.parse_args()

num_blocks = 1
hashfun = False
argdict = {}

# Check that a flag is provided
numflags = 0
if args.fullhash:
    numflags +=1
    hashfun = hashFull
else:
    if args.firstn:
        numflags +=1
        num_blocks = args.firstn
        hashfun = hashFirstn
    if args.lastn:
        numflags +=1
        num_blocks = args.lastn
        hashfun = hashLastn
    if args.firstlastn:
        numflags +=1
        num_blocks = args.firstlastn
        hashfun = hashFirstLastn
    argdict["num_blocks"] = num_blocks

if numflags != 1:
    print "Please provide a single hash flag (--fullhash (-h), --
   firstn (-fn), --lastn (-ln) OR --firstlastn (-fn))"
    exit()

print "{} {}, threads: {}".format(hashfun.__name__, num_blocks, args.
   threads)
tpool = ThreadPool(args.threads)

# Acquire file list
t0 = time.time()
flist = []
flist = os.listdir(args.dir)
t1 = time.time()
filelisttime = t1-t0
print "File enumeration time: {}, no. files: {}".format(filelisttime,
   len(flist))
if args.subset:
    flist = flist[:args.subset]
    print "Using subset of {} files".format(len(flist))

readbytes = BLOCKSIZE * num_blocks
t0 = time.time()
for im in flist:
    impath = os.path.join(args.dir, im)
    tpool.add_task(hashfun, impath, readbytes)
tpool.wait_completion()
# timer end
t1 = time.time()
total_time = t1-t0

print "Processing time: {}".format(total_time)
```

Listing E.1 Benchmark for block based sub-file signatures.

E.1.3 Hashing Functions

The following code breaks down the various sub-file hashing strategies into individual functions. Sub-file approaches catch seek exceptions instead of requesting the file size from the OS. This is faster as few files should be in the 4–12KiB range, and saves a lot of time on networked devices.

```
# Python 2
import hashlib
import os

```

```python
def hashFull(fpath):
    try:
        fil = open(fpath, 'rb')
        d=fil.read()
        h = hashlib.sha256(d).hexdigest()
    except Exception, e:
        print "Error fullhashing",fpath, e
    finally:
        if "fil" in locals():
            fil.close()
    return 1

def hashFirstn(fpath, readbytes):
    try:
        fil = open(fpath, 'rb')
        d = fil.read(readbytes)
        h = hashlib.sha256(d).hexdigest()
    except Exception, e:
        print "Error in firstn", fpath, e
    finally:
        if "fil" in locals():
            fil.close()
    return 1

def hashLastn(fpath, readbytes):
    try:
        fil = open(fpath, 'rb')
        try:
            fil.seek(-readbytes, 2)
        except:
            # didn't work, file will be too small
            pass
        d = fil.read(readbytes)
        h = hashlib.sha256(d).hexdigest()
    except Exception, e:
        print "Error in lastn", fpath, e
    finally:
        if "fil" in locals():
            fil.close()
    return 1


def hashFirstLastn(fpath, readbytes):
    try:
        fil = open(fpath, 'rb')
        try:
```

```
      # first block
            d = fil.read(readbytes)
      # last block
            fil.seek(-BLOCKSIZE, 2)
            d += fil.read(readbytes)
        except:
            # fallback, read the entire file as it's probably too small.
            fil.seek(0)
            d=fil.read()
        # hash together
        h = hashlib.sha256(d).hexdigest()
    except Exception, e:
        print "Error in hashfirstandlast", fpath, e
    finally:
        if "fil" in locals():
            fil.close()
    return 1
```

Listing E.2 Hash functions for generating block based sub-file signatures.

Appendix F

Thumbnail - Supplementary

F.1 Code Snippets

F.1.1 Generate and Fetch Thumbnails (Partial)

Below is a sample of the code use to trigger Windows 10 thumbnail generation and trigger batch extraction. Only partial code is provided, see the Github repository [149] for complete code. Timed benchmark results are found in Section 5.5.4. The full Visual Studio project allows for the thumbnail extraction executables to be built for both directly for Windows 10, and with a legacy option for Windows 8.1 and below.

```
{
        LPWSTR szChildName = nullptr;
        pChildItem->GetDisplayName(SIGDN_NORMALDISPLAY, &szChildName);

        // Extract the thumbnail from the original via the Windows Shell API
        thumbhr = cache->GetThumbnail(pChildItem,
          thumbsize,
          WTS_FORCEEXTRACTION,    // force extraction
          NULL, // Don't retrieve the memory mapped bitmap.
          &flags,
          &thumbid
        );
        if (SUCCEEDED(thumbhr)) {
          // Get CacheID ("Cache Entry Hash" in ThumbCache Viewer).
          // Viewer reverses the bytes and ignores the half which is comprised of zeroes.
          // Output here is set to match the viewer.

          stringstream id;
          id << "0x";
        for (int i = 7; i>-1; i--) {
          id << setw(2) << setfill('0') << hex << (int)thumbid.rgbKey[i];
        }

        idmap[id.str()] = 1;
        nameidmap[id.str()] = szChildName;
        if (idmap.size() == batchsize){
          i = i + batchsize;
          cout << "Cached: " << i << ", parsing thumcache.db ...";
          // Parse thumbcache and save entries which have IDs matching those in idmap to output_path
          // exportThumbs imported from another file.
          idmap = exportThumbs(dbname, output_path, idmap, nameidmap);
          // Output if any items failed to be retrieved, shouldn't happen for batch sizes << thumbcache max size.
          if (idmap.size() > 0) {
            cout << "failed to save: " << idmap.size() << endl;
          }
          else {
            cout << "success" << endl;
          }


          for (auto kv : idmap) {
            cout << kv.first << endl;
            failedIds.push_back(kv.first);   //keep track of failed items
          }
          idmap.clear(); // reset the IDs to get from the cache, these one's arent there.
          nameidmap.clear(); // reset id-filename map.
        }


      } else {
        wprintf(L"Failed to obtain %s\n", szChildName);
      }

      CoTaskMemFree(szChildName);
      pChildItem->Release();
    }
```

Listing F.1 Snippet to populate thumbnail and perform batch extraction. Only partial program.

F.1.2 Dropbox Thumbnail Benchmarks - SDK

The Python code below allows for the execution and timing of various Dropbox file acquisition approaches, as benchmarked in Section 5.7.2.

```

# Python 2
import json
import requests
import argparse
import time
import datetime
import hashlib
import dropbox
import os
from tokens import DROPBOX_TOKEN
from thread_utils import *


THUMBSIZES=[32,64,128,640,1024]
#THUMBSIZES2=[32,64,128,256,320,480,640,768,1536] # second dimension,
    not all square
THUMBTYPES=["jpeg", "png"]

THUMBSIZEMAP= {
32: dropbox.files.ThumbnailSize.w32h32,
64: dropbox.files.ThumbnailSize.w64h64,
128: dropbox.files.ThumbnailSize.w128h128,
640: dropbox.files.ThumbnailSize.w640h480,
1024: dropbox.files.ThumbnailSize.w1024h768
}

save_dir = None


def getThumbbatch(dbx, filelist, thumbtype, thumbsize):
    # filelist is listof filemetadata
    entrieslist = []
    for f in filelist:
        # path, format, size, mode
        e = dropbox.files.ThumbnailArg(path=f.path_lower,format=
    thumbtype, size=thumbsize)
        entrieslist.append(e)
    r = dbx.files_get_thumbnail_batch(entries=entrieslist)
    c = 0
    for e in r.entries:
        if e.is_success():
            c+=1
            edata = e.get_success()
            if save_dir:
                thumb = edata.thumbnail
                meta = edata.metadata
                savepath = os.path.join(save_dir, meta.name)
                try:
                    with open(savepath, "wb") as f:
                        f.write(thumb.decode("base64"))
                except Exception as err:
                    print "failed to save {}\n{}".format(meta.name, err)
    if c != 25:
        print "Got {}".format(c)


def getSingleFile(dbx, fpath, statusError):
    r = dbx.files_download(fpath)
    filemeta = r[0]
    response = r[-1]
    # Catch status code errors
    if response.status_code not in [200, 206]:
        statusError["code"] = response.status_code
        statusError["headers"] = response.headers
        print "Error on", filemeta.path_lower, response.status_code, response.headers
        return
    # file content
    thumb = response.content
    if save_dir:
        savepath = os.path.join(save_dir, filemeta.path_lower)
        try:
            with open(savepath, "wb") as f:
                f.write(thumb.decode("base64"))
        except Exception as err:
            print "failed to save {}\n{}".format(meta.name, err)

def getSingleThumb(dbx, fpath, thumbtype, thumbsize, statusError):
    r = dbx.files_get_thumbnail(path=fpath, format=thumbtype, size=thumbsize)
    filemeta = r[0]
    response = r[-1]
    # Catch status code errors
    if response.status_code not in [200, 206]:
        statusError["code"] = response.status_code
        statusError["headers"] = response.headers
        print "Error on", filemeta.path_lower, response.status_code, response.headers
        return
    # file content
    thumb = response.content
    if save_dir:
        savepath = os.path.join(save_dir, filemeta.path_lower)
        try:
            with open(savepath, "wb") as f:
                f.write(thumb.decode("base64"))
        except Exception as err:
            print "failed to save {}\n{}".format(meta.name, err)



# Set up args
parser = argparse.ArgumentParser(description="""Utility to benchmark dropbox times to fetch files and thumbnails.
""")
#parser.add_argument('alg', help="")
parser.add_argument('folderpath', help="dropbox folder path, relative to dropbox root")
parser.add_argument('--zip', '-z', action='store_true', help='Uee download_zip method.')
parser.add_argument('--singlefiles', '-sf', action='store_true', help='Uee download all files individually.')
parser.add_argument('--singlethumbs', '-st', action='store_true', help='Uee download all files individually.')
parser.add_argument('--thumbbatch', '-tb', action='store_true', help='Use thumbload_batch method.')
parser.add_argument('--thumbsize', '-ts', type=int, help='Dimension of thumbnails, Options: {}'.format(THUMBSIZES))
parser.add_argument('--thumbtype', '-tt', type=str, help='Type of thumbnail, PNG or JPEG.')
parser.add_argument('--threads', '-t', type=int, help='Thread count for multiple requests.')
parser.add_argument('--subset', '-sub', type=int, help='Maximum number of files to process.')
parser.add_argument('--progresscount', '-c', type=int, help='Print progress updates after processing this many items.')
parser.add_argument('--savedir', '-dir', type=str, help='Save files to a directory.')


args = parser.parse_args()
```

```python
# Check that thread count is provided

if (args.thumbbatch + args.zip + args.singlefiles + args.singlethumbs) > 1:
    print "Please choose either --zip (z), --thumbbatch (-tb), --singlethumbs (-st), or --singlefiles (-sf)"
    exit()

# Setup variables for thumbnail stuff
if (args.thumbbatch or args.singlethumbs):
    m = "Please provide a valid thumbnail size:\nOptions:{}".format(THUMBSIZES)
    if args.thumbsize:
        if args.thumbsize not in THUMBSIZES:
            print m
            exit()
    else:
        print m
        exit()
    # get thumbsize object from int
    thumbsize = THUMBSIZEMAP[args.thumbsize]

    m = "Please provide a valid thumbnail type:\nOptions:{}".format(THUMBTYPES)
    if args.thumbtype:
        if args.thumbtype.lower() not in THUMBTYPES:
            print m
            exit()
    else:
        print m
        exit()
    # get thumbnailformat obejcts from strings
    if args.thumbtype.lower() == "jpeg":
        thumbtype = dropbox.files.ThumbnailFormat.jpeg
    elif args.thumbtype.lower() == "png":
        thumbtype = dropbox.files.ThumbnailFormat.png

if not args.zip:
    if not args.threads:
        print "Please provide a thread count (--threads int)"
        exit()

# Set number of items to add before providing progress feedback
if args.progresscount:
    progresscount = args.progresscount
else:
    progresscount = 250

if args.savedir:
    save_dir = args.savedir
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
        print "Created save directory."
    else:
        if not os.path.isdir(save_dir):
            print "{} is not a directory".format(save_dir)
            exit()





# Check that a flag is provided
error = ThreadSafeDict()

print args
print "Start time (before connecting):",str(datetime.datetime.now())
    [:19]

# Set up access to Dropbox account
dbx = dropbox.Dropbox(DROPBOX_TOKEN, timeout=120)
dbx.users_get_current_account()


if args.zip:
    # Download zip benchmark
    # note: set timeout on dropbox.Dropbox() object for this to work
    print "downloading zip for: {}".format(args.folderpath)
    t0 = time.time()
    r = dbx.files_download_zip(args.folderpath)[-1]
    d = r.content
    t1 = time.time()


# not a standalone call to the directory. need ot fetch file paths
else:
    print "fetching list of files"
    te0 = time.time() # enumeration timer start
    # Start listing files in benchmark directory
    flimit = 2000 # max allowed for files_list_folder
    if args.subset > flimit:
        flimit = args.subset

    lf = dbx.files_list_folder(args.folderpath, limit=flimit)
    entries = lf.entries
    if args.subset:
        if len(entries) >= args.subset:
            entries = entries[:args.subset]
        else:
            # If the list spans more than a single request, keep getting them until done.
            while lf.has_more:
                lf = dbx.files_list_folder_continue(lf.cursor)
                entries.extend(lf.entries)
                if len(entries) >= args.subset:
                    entries = entries[:args.subset]
                    break

    else:
        #continue fetching all items
        while lf.has_more:
            lf = dbx.files_list_folder_continue(lf.cursor)
            entries.extend(lf.entries)
    te1 = time.time() # enumeration timer end
    enumtime = te1-te0
    numentries = len(entries)
    print "file listing time: {}".format(enumtime)
    print "num entries:",numentries

    # init thread stuff
    count = 0
    tpool = ThreadPool(args.threads)


    if (args.singlefiles or args.singlethumbs):
        # Do single files benchmark batch benchamrk
        if args.singlefiles:
            print "Starting single files benchmark"
        elif args.singlethumbs:
            print "Starting single thumbnails benchmark"
        t0 = time.time() #start benchmark timer
        for e in entries:
            if "code" in error:
                if "Retry-After" in error["headers"]:
                    timetowait = error["headers"]["Retry-After"]
                    print "Waiting on retry-after ({}) before submitting new jobs".format(float(timetowait)*1.5)
                    time.sleep(float(timetowait)*1.5)
```

```
                 error.clear()
             else:
                 error.clear()
         try:
             if args.singlefiles:
                 tpool.add_task(getSingleFile, dbx, e.path_lower, error)
             elif args.singlethumbs:
                 tpool.add_task(getSingleThumb, dbx, e.path_lower, thumbtype, thumbsize, error)
         except Exception as expt:
             print "Exception", expt, e
         count+=1
         if count % progresscount == 0:
             print "Added: ", count
     tpool.wait_completion()
     t1 = time.time() # end benchmark timer singlefiles

 if args.thumbbatch:
     # Do thumbnail batch benchamrk
     print "Starting Thumbnail_batch benchmark"
     t0 = time.time() #start benchmark timer
     i = 0
     if numentries > 25:
         j = 25
     else:
         j = numentries
     while j <= numentries:
         elist = entries[i:j]
         try:
             tpool.add_task(getThumbbatch, dbx, elist, thumbtype, thumbsize)
         except Exception as expt:
             print "Exception", expt, elist
         if j < numentries:
             i+=25
             j+=25
             if j > numentries:
                 j = numentries
             if j % progresscount == 0:
                 print "Added: ", j
         else:
             #done
             break

     print "Finished adding: {}".format(j)
```

```
        tpool.wait_completion()
        t1 = time.time() # end benchmark timer thumbnail batch


fetch_time = t1-t0
print "Fetch time: {}".format(fetch_time)
print "End time (complete):", str(datetime.datetime.now())[:19]
```

Listing F.2 Benchmark for various file acquisition approaches on Dropbox.

F.1.3 Dropbox Thumbnail Download Zip - Python Requests

A small Python program to access the download_zip Dropbox API endpoint, as the Python SDK timed out when attempting to download a large directory. Benchmark results in Section 5.7.2.

```
# Python 2
import requests
import json
import time
from tokens import dropbox_token

url = "https://content.dropboxapi.com/2/files/download_zip"

headers = {
    "Authorization": "{}".format(dropbox_token),
    "Dropbox-API-Arg": "{\"path\":\"/Datasets/Flickr_5k_intorder\"}"
}

t0 = time.time()
r = requests.post(url, headers=headers)
t1 = time.time()
print "Time taken: {}".format(t1-t0)
```

Listing F.3 Benchmark downloading Dropbox directory as Zip.

Glossary

DCT Discrete Cosine Transformation, it expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. It is an essential part of JPEG compression, transforming spatial image data into the frequency domain..

DWT Discrete Wavelet Transform. Similar in idea to the DCT, it uses a different function to transform data to the frequency domain..

JPEG This is the de-facto format for compressed images at the time of writing, being used widely on the Web and mobile devices..

Perceptual Hashing A non-cryptographic hashing technique which derives signatures from how a file appears visually to a human, rather than its binary features. Similar images should generate identical or similar perceptual hashes..

PNG Portable Network Graphics, a file format for lossless image compression..

Reverse Image Search Where an image is used as a search query resulting in a list of images with similar visual characteristics as the response, i.e. both the query and results are images, in contrast to standard text search queries..