Investigation Into A Digital Forensics Triage Tool Using Sampling, Hashes And Bloom Filters - SHAFT

John Clayton

Submitted in partial fulfilment of the requirements of Edinburgh Napier University for the Degree of MSc Advanced Security and Digital Forensics

School of Computing

20 August 2012
Authorship Declaration

I, John Clayton, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained informed consent from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Date:

Matriculation no: 95030123
Data Protection Declaration

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.
Acknowledgements

I would like to thank my supervisor, Rich Macfarlane, for his enormous assistance and guidance, for providing ideas and suggestions for improvement and for his constant encouragement.

I would also like to thank Professor Bill Buchanan, who planted the idea for this work, for his inspiration and example, and for being my internal supervisor.
Table of Contents

List of Tables .......................................................................................................................... 8
List of Figures .......................................................................................................................... 9
Abstract .................................................................................................................................... 11
1 Introduction ............................................................................................................................ 12
  1.1 Context ................................................................................................................................ 12
  1.2 Aims and Objectives ........................................................................................................... 12
  1.3 Background ......................................................................................................................... 13
    1.3.1 Operating system ...................................................................................................... 13
    1.3.2 Disk structure .......................................................................................................... 14
    1.3.3 File system .............................................................................................................. 14
  1.4 Thesis Design .................................................................................................................... 14
2 Literature Review ................................................................................................................... 15
  2.1 Introduction ....................................................................................................................... 15
  2.2 Digital Forensics Investigation Process ........................................................................... 15
    2.2.1 Data Collection ...................................................................................................... 16
    2.2.2 Data Examination .................................................................................................. 17
  2.3 Storage Capacity ................................................................................................................. 17
  2.4 Live and Dead Forensics ................................................................................................... 18
  2.5 Cloud Forensics ................................................................................................................ 19
  2.6 File Hashes ........................................................................................................................ 19
    2.6.1 Small Block File Hash Analysis ............................................................................. 19
    2.6.2 Discriminators and File Types ............................................................................ 20
  2.7 Matching Hashes ............................................................................................................... 20
    2.7.1 Bloom Filters ....................................................................................................... 20
    2.7.2 Use of Bloom Filters in IT ................................................................................ 23
  2.8 Anti-forensics and Hiding Files ......................................................................................... 23
  2.9 Sampling ............................................................................................................................. 24
    2.9.1 Sampling Files on Disk ......................................................................................... 24
    2.9.2 File Fragmentation ............................................................................................... 26
  2.10 Triage .................................................................................................................................. 27
    2.10.1 Triage in Digital Forensics ................................................................................ 27
  2.11 Taxonomy for Digital Forensics Triage ........................................................................... 27
    2.11.1 Scenarios ............................................................................................................. 27
    2.11.2 Data Size .............................................................................................................. 28
    2.11.3 Time Window ..................................................................................................... 28
    2.11.4 Taxonomy Based on Speed ................................................................................ 28
  2.12 Corpora .............................................................................................................................. 28
  2.13 Cloud-based Virtualized Environments .......................................................................... 29
  2.14 Conclusion ....................................................................................................................... 29
3 Design ..................................................................................................................................... 31
  3.1 Introduction ....................................................................................................................... 31
    3.1.1 SHAFT ................................................................................................................. 31
  3.2 Overview ............................................................................................................................. 31
  3.3 Design Methodology ......................................................................................................... 32
  3.4 Program Design ................................................................................................................ 32
    3.4.1 Make Database and Bloom Filter ....................................................................... 33
    3.4.2 Use Database for Matching Hashes ..................................................................... 33
    3.4.3 Use Bloom Filter for Matching Hashes ................................................................. 34
    3.4.4 Use Bloom Filter for Matching Hashes and Record Results ............................. 35
  3.5 Experiment Design ............................................................................................................ 36
    3.5.1 A Scientific Method ............................................................................................ 36
    3.5.2 Base Lining Experiments ..................................................................................... 37
9 Appendices.......................................................................................................................... 94
  9.1 Corpora............................................................................................................................. 94
  9.2 Code Listings..................................................................................................................... 95
    9.2.1 Makebf.py.................................................................................................................. 95
    9.2.2 Usedb.py................................................................................................................... 96
    9.2.3 Usebf.py................................................................................................................... 97
    9.2.4 Usebfmetrics.py....................................................................................................... 98
List of Tables

Table 1: Sampling - predicted hits based on sampling interval and file size.................................23
Table 2: Digital forensics triage taxonomy.................................................................26
Table 3: Bloom filter false positives - predicted vs actual (30MB).................................................40
Table 4: Bloom filter false positives - predicted vs actual (256MB)..............................................40
Table 5: Hardware specification - local PC vs remote PC............................................................48
Table 6: Baseline experiment (IDLE) 256MB - Database match vs Bloom filter match...............50
Table 7: Baseline experiment (command line) 256MB - Database match vs Bloom filter match.................................52
Table 8: Baseline experiment (command line) 256MB - IDLE vs command line.........................53
Table 9: Baseline experiment (command line) 256MB - DD disk image vs attached USB device.................................................................................................................53
Table 10: Baseline experiment (command line) 256MB - Disk times vs Match times...............54
Table 11: Baseline experiment (command line) 256MB - dd times and md5sum times..............54
Table 12: Baseline experiment (command line) 256MB - Python vs Cython...............................55
Table 13: Baseline experiment (command line) 256MB - Local PC vs remote PC..............................55
Table 14: Baseline experiment (command line) 256MB - comparison local PC vs remote PC.................................................................56
Table 15: Baseline experiment (command line) 256MB - Remote PC extended crontab execution.................................................................................................................56
Table 16: Baseline experiment (command line) 256MB - second comparison local PC vs remote PC.................................................................................................................56
Table 17: Baseline experiment (command line) 256MB - second base lining sanity check...........59
Table 18: 256MB - Comparison of various Bloom filter m and k values – program times...........60
Table 19: 2GB - Comparison of various Bloom filter m and k values – program times..............60
Table 20: 256MB - Comparison of various Bloom filter m and k values – match times.............62
Table 21: 2GB - Comparison of various Bloom filter m and k values – match times.................62
Table 22: 256MB various Bloom filter values – True positives and accuracy..............................66
Table 23: 2GB various Bloom filter values – True positives and accuracy.................................66
Table 24: 256MB - Comparison of various fragmentation – True positives and accuracy.........68
Table 25: 2GB - Comparison of various fragmentation – True positives and accuracy.............69
List of Figures
Figure 1: Directories and files – Linux GUI view.................................................................11
Figure 2: Directories and files - Linux CLI view.................................................................11
Figure 3: Disk structure (Microsoft, n.d.)........................................................................12
Figure 4: Digital crime scene investigation phases (Carrier & Spafford, 2003)...............14
Figure 5: Data collection process - example from USB flash drive.........................15
Figure 6: Example of Bloom filter false positive rates (Roussev et al., 2006).........21
Figure 7: File fragmentation study (S. L. Garfinkel, 2007)........................................24
Figure 8: Design Overview - scenario.............................................................................29
Figure 9: Design Overview - Process.............................................................................30
Figure 10: Waterfall model (El-Haik & Shaout, 2010)..................................................30
Figure 11: Program design - Make database and Bloom filter..............................31
Figure 12: Program design - Use database...................................................................32
Figure 13: Program design - Use Bloom filter.............................................................33
Figure 14: Program design - Use Bloom filter with metrics.........................................34
Figure 15: General experiment program flow.................................................................35
Figure 16: Disk image files - overview of structure showing fragmentation.............38
Figure 17: IDLE - Python development environment..................................................41
Figure 18: Program implementation - Make database and Bloom filter..................43
Figure 19: Program implementation - Use database.....................................................44
Figure 20: Program implementation - Use Bloom filter.............................................45
Figure 21: Program implementation - Use Bloom filter with metrics.......................47
Figure 22: IDLE - Average times vs sampling interval - Database match vs Bloom filter match..........................................................51
Figure 23: IDLE - Average times vs cycles - Database match vs Bloom filter match....51
Figure 24: Command line - Average times vs sampling interval - Database match vs Bloom filter match..........................................................52
Figure 25: Command line - Average times vs cycles - Database match vs Bloom filter match..........................................................52
Figure 26: Baseline experiment (command line) 256MB - remote PC program times vs time of day..........................................................57
Figure 27: Baseline experiment (command line) 256MB - second comparison remote PC program times vs time of day..........................................................57
Figure 28: 256MB various Bloom filter values – Av. program time vs sampling interval.....58
Figure 29: 2GB various Bloom filter values – Av. program time vs sampling interval.....61
Figure 30: 256MB various Bloom filter values - Min program time vs sampling interval.....61
Figure 31: 2GB various Bloom filter values - Min program time vs sampling interval.....61
Figure 32: 256MB various Bloom filter values - Repeat av. program time vs sampling interval ..........................................................61
Figure 33: 256MB various Bloom filter values – average match time vs sampling interval...63
Figure 34: 2GB various Bloom filter values – average match time vs sampling interval...63
Figure 35: 256MB various Bloom filter values – Min. match time vs sampling interval.....63
Figure 36: 2GB various Bloom filter values – Min. match time vs sampling interval.....63
Figure 37: 256MB various Bloom filter values – Accuracy vs. average match time ........64
Figure 38: 256MB various Bloom filter values – Hit rate vs average match time ...........64
Figure 39: 256MB various Bloom filter values – Hit rate vs average match time ...........65
Figure 40: 256MB various Bloom filter values – Hit rate vs average prog. time ..........65
Figure 41: 256MB various Bloom filter values – True positive hit rate vs. sampling interval 66
Figure 42: 256MB various Bloom filter values – Accuracy vs. sampling interval ............66
Figure 43: 2GB various Bloom filter values – True positive hit rate vs. sampling interval .......67
Figure 44: 2GB various Bloom filter values – Accuracy vs. sampling interval ............67
Figure 45: 256MB - Comparison of various fragmentation – True positives hit rate vs sampling interval............................................................................70
Figure 46: 2GB - Comparison of various fragmentation – True positives hit rate vs sampling interval.................................................................70
Figure 47: 256MB - Comparison of various fragmentation – Accuracy vs sampling interval.70
Figure 48: 2GB - Comparison of various fragmentation – Accuracy vs sampling interval.....71
Figure 49: 256MB - Comparison of various fragmentation – True positive hit rate vs. accuracy......................................................................................71
Figure 50: 2GB - Comparison of various fragmentation – True positive hit rate vs. accuracy 72
Figure 51: 256MB - Comparison of various fragmentation – Ratio predicted/actual hit rate (based on fragment size) vs. sampling interval.................................................................72
Figure 52: Figure 39: 256MB - Comparison of various fragmentation – Ratio predicted/actual hit rate vs. sampling interval.................................................................72
Figure 53: 2GB - Comparison of various fragmentation – Ratio predicted/actual hit rate (based on fragment size) vs. sampling interval.................................................................73
Figure 54: 2GB - Comparison of various fragmentation – Ratio predicted/actual hit rate vs. sampling interval.................................................................73
Figure 55: Project plan GANTT chart.................................................................................................................................89
Figure 56: Project diary - week 8.................................................................................................................................90
Figure 57: Project diary - week 9.................................................................................................................................91
Figure 58: Project diary - week 10.................................................................................................................................92
Abstract

There is a problem in the world of digital forensics. The demands on digital forensic investigators and resources will continue to increase as the use of computers and other electronic devices increases, and as the storage capacity of these devices increases. The digital forensic process requires that evidence be identified and examined, and resources to do this are constrained. This is creating a backlog of work as seized media and devices wait to be analysed, and some investigations or checks 'in the field' may be reduced or discarded as impractical. There is a technique which can be used to help quickly to collect and examine data to see if it is of interest. This technique combines statistical sampling and hashes. This thesis describes the design, implementation and evaluation of a prototype digital forensics investigation triage tool to use sampling, disk sector hashing and Bloom filters.

The design of the tool and of an experiment environment to assess the tool is described. This tool can use a database or Bloom filter to match the hashes from disk sectors against the stored hashes for a file which is being searched for. The design also covers a program to prepare the database and Bloom filter which will be used for the hash matching.

The tools were successfully implemented in two separate but functionally identical experiment environments. Development and initial experiment was done on a local home PC. Further experiment was carried out on a remote virtual PC for better performance. The tools were successfully implemented and executed in both environments.

In the evaluation, the program generated both database and Bloom filter for the file. Using the tools the file was located when searching the source disk image file, and when sampling the detection rate increased with increased sampling. The Bloom filter false positive rate was as predicted by theory (Roussev, Chen, Bourg, & Richard, 2006) which confirmed that the Bloom filter had been correctly implemented. Base lining was successful in establishing the experiment environments and the effect of environmental factors on performance. Repeated experiments by program loops and scheduling the programs to run repeatedly using crontab were successful in producing a body of experiment results which were written to file for analysis. These experiments succeeded in analysing the effect of varying Bloom filter parameters and of file fragmentation.

Metrics for experiment evaluation included overall program time, match time, and disk read and hashing time. Metrics were recorded for Hit rate which is a measure of the proportion of experiments with a set of parameters which found any match with the file. Hit rate is measured over a series of experiments and is a probability with a maximum value of one. Metrics were also recorded for Accuracy which measures the ratio of true positive matches to all positive matches. Accuracy is a ratio with a maximum value of one which would mean all matches were true positives. Any value for Accuracy less than one means that there were some false positives. From the evaluation it was established that the choice of Bloom filter bit length was significant in the number of false-positives and that this also had a noticeable effect on the match time. The choice of the number of Bloom filter hashing algorithms had little effect on performance. File fragmentation had no significant effect on performance.

The dissertation confirms the successful proof of concept of working tool for digital forensics investigation triage using sampling, hashing and Bloom filter which works with a false positive rate which can be predicted. This tool was written in Python which proved a simple to use programming language. This prototype tool can provide the basis for further work on a practical tool for use in real world digital forensics investigation. The dissertation concludes with ideas for possible future work, and a personal reflection.
1 Introduction

1.1 Context

In 2009, Deputy Assistant Commissioner Janet Williams, the lead for ACPO (the Association of Chief Police Officers) on e-crime described the backlog in analysing seized data as “as one of the biggest e-crime problems”. (Blincoe, 2009). Others have also identified the problem caused by increasing amounts of suspect data which needs forensic analysis (Iii & Orleans, 2006). When describing the problems with backlog in analysing seized data Deputy Assistant Commissioner Janet Williams also stated that in 2010 it was planned to introduce a forensics triage tool to solve the problem (Blincoe, 2009). It is not clear if she was aware that in 2007 Nottinghamshire Police began using such a tool, Triage Examiner, and by 2011 about half UK police forces were using it (Anon, 2012a). Harry Parsonage, then a Nottinghamshire officer introduced the software and now works for the supplier, ADF Solutions (Anon, 2012b). There is no publicly available independent assessment of how well the ADF tool functions. This is problematic in evaluating its performance and assessing possible alternatives. This lack of independent testing of forensic tools in general is something which others have identified (Flandrin, 2012) (P. W. J. Buchanan, Macfarlane, et al., 2011) (W. Buchanan & Macfarlane, 2011) and have proposed what would be a useful standard platform for comparing forensic tools. The ACPO ((ACPO, n.d.-a) also recognises the problem of a backlog of work and that triage can assist in identifying which suspect media or devices merit further investigation.

Search techniques for files have long made use of file hashes. Extending this technique sector hashes can be searched for, that is hash values for individual distinct sectors from a known file. Matching these hashes can be done using a database lookup or a Bloom filter lookup. The Bloom filter offers speed advantages with a predictable rate of false positives. Bloom filters can be used to represent large datasets, and can quickly be used to check if a value, such as a file hash is possibly in the datasets.

By sampling sectors from a disk, rather than reading the whole disk, it is possible to locate files with predictable accuracy but in a fraction of the time. The probability of locating a sectors from a file of a give size on a disk can be calculated. As this is not a full disk scan then the probability of locating the file can be less than 1 but as part of a digital forensics triage process this method can speed up and enable investigations which would otherwise falter or stop. This triage method can identify media or devices for further in-depth investigations.

Combining the sampling of sectors from a disk and using a Bloom filter to check if the file sector hash matches the file sector hash for a known file has been described as method which could be useful (S. Garfinkel, Nelson et al 2010).This thesis describes the implementation of a tool to implement sector hash matching, with sampling and using a Bloom filter. The tool is written in Python and the source code is included. The experiment process is described, including discussion on the need for a scientific method. The successful implementation and evaluation of a proof of concept tool is described. There have been other research implementations of this method for locating files on a disk, and it seems odd that the technique is not widely and routinely adopted and freely in use.

1.2 Aims and Objectives

The aim of this thesis is to investigate digital forensics investigation triage using the theory behind sampling small block file for hash analysis, and to experimentally test that theory, and investigate the effects of changing the Bloom filter settings and the effect of file fragmentation. Objectives to support this aim are as follows;

1 Investigate the issues in digital forensics investigation triage relating to data
collection and examination, in particular around locating files by using sector sampling, hashes, and Bloom filters.

2 Design digital forensics investigation triage tools and design experiments to investigate the effect of varying Bloom filter settings and of file fragmentation on the performance of the tools.

3 Implement prototype tools and an experiment environment. These tools are to include using database and Bloom filter matching.

4 Evaluate and assess the results of the experiment on the tools to assess the effect of varying Bloom filters and of file fragmentation, and identify possible future work.

1.3 Background

This section introduces the fundamentals of how digital information is stored on a computer disk. This is basic to understanding how the digital forensics investigation triage tools work.

1.3.1 Operating system

Computers provide a means to input, store, process and analyse data. Users are familiar with the concept of files and directories (or folders) being used to store data. The operating system displays the data structures to the user in an easily accessible form. Ubuntu Linux is typical, and files and directories can be displayed graphically, as follows;

Figure 1: Directories and files – Linux GUI view

Files and directories can also be displayed at the command line as follows;

Figure 2: Directories and files - Linux CLI view

Files and directories can be selected, moved, copied and opened. If a file is opened it is
opened by an application, so that for example a html page can be opened in a web browser. If a directory is opened then the files and directories contained within it are shown. This is familiar to most users and hides the user from the physical storage of data.

### 1.3.2 Disk structure

The operating system presents a representation of stored data as an interface for the user and applications. It is common for data to be stored on a hard disk of rotating platters. Each platter is organised by the file system into tracks, cylinders and sectors. A sector is the smallest addressable storage unit on a disk (Carrier, 2005).

![Disk structure diagram](image)

*Figure 3: Disk structure (Microsoft, n.d.)*

### 1.3.3 File system

The operating system uses a file system to access data on the disk, for instance Windows NT uses NTFS (New Technology File System). The file system organises sectors into clusters. A cluster is the smallest storage unit addressable by the file system (Morley & Parker, 2012). Files can span several sectors which may be continuous or fragmented across the disk.

### 1.4 Thesis Design

Chapter 2 explores the literature on digital forensics investigation process. The chapter then investigates issues which show the requirement for digital forensics investigation triage, and the techniques of small block hashing, Bloom filter and sampling which can be combined in a tool.

Chapter 3 provides a design for a digital forensics investigation triage tool and the design of experiments to assess it, including the effect of varying Bloom filter parameters and of file fragmentation.

Chapter 4 describes the implementation of the digital forensics investigation triage tool and the experiment environment.

Chapter 5 evaluates and analyses the results of base lining the experiment environment and the results of the experiments on the digital forensics investigation triage tool.

Chapter 6 concludes the thesis, and includes ideas for possible future work.
2 Literature Review

2.1 Introduction

This chapter researches into several topics which are relevant to the aims and objectives of the thesis. There is a process to digital forensics investigation, and that process includes data collection and examination (Reith, Carr, et al., 2002). The collection and examination of data is affected by several factors, such as the increase in storage capacities, (Eleftherious & Al., 2011) the need for live forensics (Iii & Orleans, 2006), and the use of cloud based systems. (Birk, 2011) (Kaufman, 2009) A range of techniques can be applied to this collection and examination including the use of file hashes to search for files (Whitman & Mattord, 2011). Matching of these file hashes can use Bloom filters (Farrell, Garfinkel, et al., 2008) and sectors can be sampled from a disk instead of full disk analysis. This can also help defeat anti-forensics and locate hidden files. Combining these techniques can support digital forensics triage, (Cantrell, Dampier, Dandass, et al., 2012) (Overill & Silomon, n.d.) which is applicable in a range of scenarios. These scenarios can be used to support a taxonomy for this triage. The reliable testing of a tool to use these techniques can use digital corpora. (Woods et al., 2011), As well as a literature review, it is useful to describe two concepts in particular, one of which 'statistical sampling' will probably be familiar to readers who have studied mathematics and a second concept, 'Bloom filter', which may be less well known.

2.2 Digital Forensics Investigation Process

'Forensic' means 'relating to or denoting the scientific investigation of crime', 'belonging to or used in courts of law' (Allen, 2004). So forensic investigation must be scientific, and also satisfy court standards for evidence. This applies to digital forensics, as much as to other forensics, such as medical forensics. ACPOS (Association of Chief Police Officers in Scotland) for Scotland, and ACPO for the rest of the UK have guidelines to help digital forensic investigations be forensically sound (ACPO, n.d.-b). These guidelines embody four principles, namely;

1. Data which might be used in evidence should not be altered. This can be verified by the use of checksums to compare the data as seized with a data copy used for analysis.
2. Any investigator must be technically competent and have a demonstrable knowledge of digital forensics.
3. A chain of evidence must be kept to provide an audit trail, as with physical forensic evidence.
4. The investigator in charge has overall responsibility for seeing that proper procedures are followed.

During an investigation acquiring data, the use of a write blocker supports principle 1, while logs and records of analysis with findings supports principle 3.

The digital forensics process can be described as a nine step model (Reith, Carr, et al., 2002)

1. **Identification** that an incident has occurred which needs investigation. This could be a reported incident or a discovered incident, and it may be suspicion or certainty that there has been an incident.
2. **Preparation** for an incident, including the preparation of tools.
3. **Approach strategy** which is planning the collection of evidence in an optimum way.
4. **Preservation** of the crime scene. This may involve isolating the system.
5. **Collection** of devices, media, data and information required for investigation.

6. **Examination** of the evidence to determine what is relevant for further investigation.

7. **Analysis** in detail of the evidence identified for further investigation, to determine what happened.

8. **Presentation** of the findings in a forensically sound and intelligible form.

9. **Returning** evidence which can also be extended to retention or disposal of evidence if that is required.

This is one description and could be shortened, but it does cover the process. Others have also described digital forensics process models (Carrier & Spafford, 2003) (Ciardhuáin, 2004), Carrier likens the computer to the body at a murder scene using the analogy that both contain evidence of the crime. Following the analogy with physical crime scene investigation the digital crime scene phase is divided further into six components in Carrier's work.

![Digital Crime Scene Investigation Phases](image)

*Figure 4: Digital crime scene investigation phases (Carrier & Spafford, 2003)*

This thesis will be focussing on the use of a tool for the data collection and data examination steps from the nine step model (Reith et al., 2002).

### 2.2.1 Data Collection

Step 5 of the 9 step digital forensics process described above is data collection. There are several methods for acquiring data for analysis. The actual computer or disk may be seized, and typically the whole disk imaged, using a write-blocker to protect the original and generating a hash of the image for integrity purposes. This full forensics process can be time consuming. In the case of creating an image of a USB drive for instance, the Unix `dd` command can be used to do a bit copy of the disk to file. Using a write blocker stops any possibility of interfering with the original data on the USB drive. Generating a hash of the disk image provides referential integrity. Analysis can be carried out on the disk image, and after analysis the hash of the disk image can be regenerated and checked to confirm that the disk image has not been changed by the analysis. This provides a chain of evidence that the analysis was carried out on data which remained identical to the original data. This data collection process example is illustrated below.
Where time is a constraint, there is sometimes a need to do a quick analysis to see if there is evidence to support a fuller investigation. This could be for instance at a border control. The suspect medium could be a computer hard disk, or other media such as a camera SD card. For a suspect computer, the analysis could be performed by connecting another computer using a network cable, or by running forensic analysis live off attached USB (Yang & Yen, 2010). (Yen, Yang, et al, 2008) For removable media, such as a SD card, then a card reader could be used.

### 2.2.2 Data Examination

Step 6 of the 9 step digital forensics process described above is data examination. Having collected the evidence the next step is to examine the evidence to identify what needs to be investigated further. This is not the full in-depth analysis step. In the case of examining a disk for a file for instance, a range of techniques are available to locate files which may be of interest. If the file name is known the simplest is to search by name using the operating system. Other options are to search by file type, size, and folder location. The exact options to use will depend on the operating system, but file size for instance is a common option, as in Windows 7 (Stanek, 2009) and Linux (Negus & Caen, 2011). The recycle bin in Windows can be checked for deleted files (Hart-Davis, 2007). Going into more depth, hidden and fragmented files can be searched for, including for instance using hashes which are described later.

How long this examination takes will depend on several factors, including the volume of evidence and how it is to be examined. The choice of examination techniques can affect the speed and the reliability of the examination. The amounts of evidence to be analysed are increasing with increasing storage capacity.

### 2.3 Storage Capacity

The time take for data collection and examination depends on the speed at which data can be processed, and the amount of data to be processed. As the amount of data increases so does the resourcing problem for the digital forensics investigator. Digital storage capacity is increasing and with it increases the amount of data to be analysed, and traditionally to be imaged. Moore's Law is well known and states that the number of transistors and resistors on a chip doubles approximately every two years (Anon, n.d.-a). Data storage capacities are increasing at an even greater rate, from the first appearance of the hard disk drive in 1956, by 2005 the density of bits stored increased by about 50 million times and this growth has been referred to as Kryder's Law (Walter, 2005). This means that the the increase in data is outpacing the increase in processing power. More devices have more storage, and more people use them, such as a smartphone, a camera or a laptop, and more criminals use them.
The rise in storage capacities clearly presents a problem for digital forensic examiners. It can delay investigations, and make routine checks for content impractical, as at border controls for example.

Over the period 1980 to 2010 hard disk drive (HDD) capacity increased at an annual rate of about 60% (Eleftherious & Al., 2011). This is a remarkable rate and is most immediately apparent in the storage capacity on familiar devices, such as PCs under £500 with Terrabyte HDD (Dell, 2012) and SD cards for smartphones and cameras with storage capacities of 32GB (Samsung, 2012). However, along with this increase in capacity read rates limit the speed at which a complete disk can be read. For instance, the Samsung 32GB SD boasts a read rate of 24MB/s, and even at that it would take over 20 minutes to read. HDD are faster, with a typical read/write rate of around 156MB/s (Seagate, 2012) but even at that rate a 1TB HDD would take over 100 minutes to read. Even if these rates can be sustained, what is then read and written needs to be stored and analysed. As trends continue, the problem in analysing storage media in a practical time will increase (Iii & Orleans, 2006).

Historically digital evidence had usually been taken from hard disk, but increasingly data will be taken from SSD (solid state drives), such as USB memory sticks, and collecting data off SSD has additional difficulties, such as the fact they can be quickly and efficiently wiped of data (Bell & Boddington, 2010). None of this will address directly the issue of cloud data storage, where data is held remotely rather than locally (Joint, Baker, et al, 2009), and this dissertation is concerned mainly with looking for data available on a local medium.

Trends suggest that data storage capacity will continue to increase, and with it the associated time take to do data analysis. Although this can be offset somewhat by faster data transfer rates and faster CPUs, these are increasing more slowly, in addition to which more devices are appearing with significant data storage capacity (smartphones, cameras), so the scale of the problem in analysing suspect media is likely to increase. The technique can alleviate this problem by accelerating analysis. Another factor which affects collecting data is whether the system is live or dead.

### 2.4 Live and Dead Forensics

As well as the problem of data storage capacity, live systems can be difficult to isolate and copy in total, and a fast process which could be applied in situ to the live system would be useful. The static, or ‘dead box’ forensics investigation has the data available in a static form. This may be the original source disk, but more usually is a bit copy which is identical for forensic investigation purposes but does not compromise the integrity of the original. There are differences between live and dead forensic investigations. With a dead box investigation it is always possible to return for further investigation, and the data can be reliably recreated. With a live investigation the system is still running. The system state is changing, and may be affected by the investigation and tests cannot be guaranteed to be repeatable. However, despite these apparent disadvantages live investigation is sometimes necessary.

There may simply not be the time or the storage capacity to image a system. It might be large and critical, such as a corporate mail server, or the system may only be briefly available for examination, such as at a border crossing. The system may be impossible to isolate or reproduce, such as a externally hosted cloud system. The increase in system capacities, including larger disk sizes is increasing the number of times it is impractical to isolate a system for dead box analysis and it must be analysed live (Adelstein, 2006).

A trend has been identified towards live forensics investigation where the suspect machine remains in operation (Iii & Orleans, 2006) and that fits the typical scenario envisaged in this dissertation, where a suspect machine or medium is examined in its live state. These techniques could also find application in a different live context with network data (Nikkel, 2005). Although network traffic sampling is a possibility for further future study using similar techniques to those applied to small block file analysis it will not be investigated here. Another factor which affects collecting data is the increasing use of cloud computing.
2.5 Cloud Forensics

Similarly to live systems, the move to cloud computing, where the environment is controlled often by a service provider, and cannot be isolated, contained or reproduced by an investigator presents a challenge for digital forensics and the investigator (Birk, 2011)(Kaufman, 2009). This makes the problem of isolating the crime scene more difficult, and maybe impossible. Even where the data is relatively small in size, knowing the extent and location of the data when accessing a virtualised system, is clearly much less straightforward than when the data is contained on a PC or removable media. The volatility and lack of control increases the need for faster analysis of data, using techniques such as triage (Rogers, Mislan, et al, 2006) as used in this dissertation. Some security researchers have characterised cloud computing as a return to centralized computing, but now in the virtual (Kandukuri, V., & Rakshit, 2009). There has been rapid growth in cloud computing, and it has even been claimed that security is the only obstacle to its wide adoption (Haider, 2011). Some researchers conclude that neither cloud providers or forensics investigators are on top of the security concerns raised by the adoption of cloud computing (Reilly et al., 2011). The distributed and remote nature of cloud computing also increases the jurisdicitional difficulties (Vaciago, 2012) which is one of the factors which increase the complexity of digital forensic investigation of cloud computing systems (Taylor, Haggerty, et al, 2010)(Taylor, Haggerty, et al, 2011).

2.6 File Hashes

Looking for a match between the data under examination requires a means to compare it with the file being looked for. The use of file hashes to 'fingerprint' files for identification purposes, and to check integrity is well known (Whitman & Mattord, 2011). A file is read, and an algorithm is used to generate a distinct large number called a hash. Given that a known bad file might be renamed or hidden on a disk, the typical approach to find it is to use file hashes, computing these for every file on the disk and comparing them. Hashes are typically generated for whole files, but can equally be generated for small block of several disk sectors or even for individual sectors. There are several hashing algorithms, e.g. MD5 (message digest algorithm) and SHA1 (secure hash algorithm) (Turnbull, 2005) which are commonly used. The hashes will only be distinct and useful for identification if the input is distinct, so when doing any hash comparison using sectors or blocks any which are null should be disregarded because this is a common value and is not distinct. Nulls can be used for instance for 'padding' (or 'stuffing') unused sectors in a cluster(Banzal, 2007). Searching for files by hash is a general technique for digital media, and is not limited to computer disks, for instance SIMCon can search SIM cards for hidden data using SHA1 hashes (Ayers & Daniellou, 2005).

2.6.1 Small Block File Hash Analysis

It has been described how the sampling of small blocks off disk and hashing can be used in forensics (S. Garfinkel et al., 2010). This research has two main themes. One is identifying files types from small blocks, looking for discriminators such as are found in jpeg file types. The other theme is looking for known files by sampling small blocks. The concept of 'distinct blocks' is introduced, although it is self evident. This is the idea that to compare the hashes of two small blocks to see if they come from the same file, then the small block must be distinct, that is not found in other files. When establishing a reference set of block hashes, where a block is a collection of sectors, then multiple hashes will be required. This is because when sampling blocks from the disk, the starting cluster is random and so if the file is in the reference set then it's starting sector (and hence hash value) must match a block in the reference set which starts at the same sector. To do this then to match a block found at random on the disk will require hashes in the reference set corresponding to any sector being the first sector, that is a number of hashes equal to the number of sectors, although each one corresponds to a block of sector.
2.6.2 Discriminators and File Types

Another use of small block hashing is to identify file types generally. As well as it's use for finding a known bad file from a sample block, the same technique of small block analysis can be used to identify file types, if these file types have distinctive patterns, 'discriminators' - which can be found within the block. The file type for complete files can be recognised by the leading bytes, the 'magic bytes', at the start of the file (Sammes & Jenkinson, 2000). For instance, the leading bytes of a jpeg file in hexadecimal are FF D8 FF E0. This works well for a complete file being analysed but does not help with identifying a file type from a sampled block which has a different starting point. Even the presence of the magic bytes as an identifier of file type has been criticised, as for instance a jpeg may be embedded within a pdf (S. Garfinkel et al., 2010). They described that much work on this has used unigram and bigram analysis and cited several papers, and there has been research on the use of long N-grams for this also (Mayer, 2011).

2.7 Matching Hashes

The hashes of the sectors read off disk need to be checked for a match against the hashes from the sectors in the search file. The search file sector hashes can be stored in a database, and this is the most obvious approach. As the sectors are read off disk, they are hashed and the database is checked for a match on that hash value. As the size of the database increases, then it would be expected that this look up will take longer. In real world examples the database could conceivably contain the individual hashes for each sector for thousands of files. So, it is worthwhile to consider whether there are efficient alternatives to a database for doing this matching.

2.7.1 Bloom Filters

As speed of analysis is an issue, then Bloom filters are of interest as they offer a fast way to make a comparison, in this case between the hash of a sector from a disk and the reference set of sector hashes for the known bad file. Bloom filters provide a means to quickly test whether an object is a member of a set. They can return a true negative, but not a false negative. They can return a true or a false positive. The eponymous Bloom filter was developed by Burton H. Bloom (Bloom, 1970). The purpose of the Bloom filter is to represent members of a set in a bit vector, which can then be used to compare an object and see if it is equal to a member of the set. Bloom filters cannot give false negatives, that is an object identified as not being a
member of the set definitely will not belong to it. The Bloom filter can give false positives, but at a predicted level of probability. The Bloom filter match does not absolutely confirm a match, but as effectively a pre-processor it reduces the amount of checking whether an object is the member of a set by eliminating many true negatives.

The Bloom filter is a single bit vector initialized to be 0 0 0 0 0 0 .. etc. It is a bit representation of the members of a set. There is a set of values, which could for example be data blocks, or dictionary words, or urls, against which objects are to be compared to see if they are in the set. Each of these values is hashed to produce a number. The hash value of this is used to set the corresponding bit in the Bloom filter to 1. There can be multiple hashes of the same value using different hash functions. This is repeated for the all the members of the set to result in a Bloom filter of the form 1 1 0 0 1 0 .. etc. where each 1 means at least one member of the set has a hash value corresponding to that index in the Bloom filter. To query if another object is possibly a member of the set, the same algorithm is used to produce hashes for this object and a comparison made with the Bloom filter. If every bit set to 1 by the object is also set to 1 in the Bloom filter, then it may be a member of the set. However, if any bit set to 1 by the object is set to 0 in the Bloom filter then it cannot be a member of the set.

Let

\[ m = \text{the number of bits in the Bloom filter} \]
\[ k = \text{the number of hashing function} \]
\[ H \text{ be the hashing function} \]

an example where \( m = 16, \ k = 3, \text{members} = \{ X0, X1,X2 \} \)

Bloom filter initialised to 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

There are 16 bits in the Bloom filter, so the hashing function \( H \) has to return a value between 0 and 15 with equal probability of any particular value. With \( m \) the number of bits in the Bloom filter, then \( H \) has to generate an index value between 0 and \( 1-m \). So if \( m \) is 16 then \( H \) has to generate values between 0 and 15 which would require \( H \) to be a 4 bit number as \( 2^4 \) will give 16 values. Accordingly, sometimes Bloom filter lengths are expressed as an exponential of 2, and that gives the bit size needed from the hash, such that a Bloom filter of length \( 2^{16} \) bits needs a 16 bit hash which will give \( 2^{16} \) possible values. As in this case there are 3 hashing functions each to produce a value between 0 and 15 to be used to set that index in the Bloom filter to 1.

e.g. \( H(X0) = \{2, 4, 6\} \), \( H(X1) = \{0,3,6\}, H(X2) = \{7,8,9\} \)

adding the hash values as a bit vector to the initialised Bloom filter

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialised Bloom filter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Hashed values of member x0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Resulting Bloom filter</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Then continue for the other members i.e.

| Hashed values x1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Resulting Bloom filter | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hashed values x1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
The final Bloom filter can then be used to see if an object may be a member of the set. If the object is a member of the set then it will have the hash values that correspond directly to set bits in the Bloom filter. If any of the hash values of the object correspond to a 0 in the Bloom filter then that proves that it cannot possibly be a member. A false negative is impossible. However, a false positive is possible. For instance in this example, and object could have hash values \{2,3,6\}. Those bits are set to 1 in the Bloom filter, however they have not all been set as a result from hashing a single member, but comprise results from hashing several members. This is a trivial example. With a small bit length a Bloom filter would soon be set to all ones, as Bloom filters are typically used to represent large sets the bit length in practice is much larger.

As well as being a compact representation, Bloom filters allow fast comparison, as a look up could be done of each of index of the bit set to 1 by the object. If any of those indices in the Bloom filter is a 0 then that is a negative, otherwise it is positive, but it may be a false positive or a true positive. As mentioned, Bloom filters can give false positives, but not false negatives. When looking if an object is a member of a set, a false negative would be of concern. However, a false positive should only mean that if the members of the set were then queried directly the false positive would become apparent. While this would have involved unnecessary processing, then how significant the false positive is would depend on the circumstances. For instance a false positive in a url cache simple requires reloading the url cache from the source.

Although the concept of the Bloom filter is clear and elegant, the mathematics involved and the calculation of false probability rates are not intuitive. The mathematics here follows from (Roussev, Chen, et al, 2006) who cite (Fan, 2000) and (Mitzenmacher, 2002).

Where, \(n\) = the number of elements in the set whose membership will be represented by the Bloom filter, \(m\) = the number of bits in the Bloom filter and \(k\) = the number of hash functions to be applied to each element and then update the Bloom filter

For the first hash of the first element the probability of any bit in the Bloom filter being set to 1 is \(1/m\), and so the probability that it remains 0 is \((1 – 1/m)\). This repeats, so that the probability that a bit remains at 0 after \(n\) elements have been hashed \(k\) times is \((1– \frac{1}{m})^kn\).

That means the probability that a bit is set to 1 is \((1–(1–\frac{1}{m})^kn)\).

When an object is hashed and compared against the Bloom Filter, the random probability that the corresponding bits are set to 1 would represent a false positive. That is the chance that a bit is set to 1 even if there is no corresponding element with the same hash as the object being checked. The probability of a false positive is then given as;

\[ P=\left(1-(1-\frac{1}{m})^kn\right) \]

which approximates to \(1-e^{-(kn/m)}\).

For a given \(m\) and \(n\), the false positive rate is minimum when \(k = \ln 2 \ (m/n)\). Increasing \(k\) will increase the chance of one of the bits generated for the object is 0, but it also increases the proportion of bits in the Bloom filter which are set to 1. In their case they give the following examples of false positive rates.
e.g. \((1 - e^{-12/16})^2 = 0.013807\)

The predicted probability for false positives can be compared with experiment results to check the validity both of the Bloom filter calculation, and its implementation in the experiment.

### 2.7.2 Use of Bloom Filters in IT

While the existence, let alone the mathematics, of Bloom filters is probably not known by many computer users, many aspects of IT do make use of Bloom filters. Bloom filters have found application within computing, such as representing the NIST RDS (Reference Data Set) of 13 million hashes of known traceable software applications (Farrell, Garfinkel, et al, 2008). It speeds up checking if an objects is in the set, in this case the NIST RDS, by representing the members compactly and in a form which can be quickly compared. Other uses of Bloom filters include network traffic (Song, Louis, et al, 2005) (Broder & Mitzenmacher, n.d.) (Xu & Wang, 2003), file hashes (Roussev et al, 2006), web caching (Fan, 2000) (Wang & Network, n.d.), multi-processing (Chung, Kozyrakis, et al, n.d.) and spell checkers (Blustein & El-Maazawi, 2002) (Kann, Domeij, et al, 1998). Using Bloom filters to store network traffic in a security context has also been described (Masti, 2010). In web caching for instance, a Bloom filter is used to see if a URL resource is cached locally. If the Bloom filter gives a false positive, then an unnecessary lookup is performed on the local cache, and the url is then retrieved remotely. This overhead is offset by the speed gain in checking urls.

The use of Bloom filters in storing hashes for forensic analysis has been described (Roussev et al., 2006). They describe using a Bloom filter to store the hashes of 4096 byte size blocks. Compared with storing the values in a hash table the Bloom filter takes up less space in memory and allows faster comparison. Bloom filters were also used by Garfinkel for hash comparisons (S. Garfinkel et al., 2010). A SHA1 hash of every block was stored in a \(2^{32}\) bit Bloom filter, with \(k\) (number of hash functions) = 4. The SHA1 hash was used to generate the 4 hash functions for the Bloom filter in an elegant way. SHA1 gives a result as 160 bit binary number. Because the hash is distinct, 32 bit sections from it are reasonably distinct also having \(2^{32}\) combinations. So, the first 32 bits are used to set one index in the Bloom filter (between 1 and \(2^{32}\)). The second 32 bits are used to set the second index, and so on. Sector sampling and hashing can help in locating hidden files.

### 2.8 Anti-forensics and Hiding Files

It may be that the file being looked for is present but is hidden in some way. While they still have some value, simple forensic tools can be defeated by anti-forensics, even by just deleting files (Gillam & Rogers, 2005). As 'deleted' files typically leave content in sectors on the disk, there are tools and techniques to recover deleted files. These include the undelete functionality of some operating systems, such as the Windows recycle bin, and also the specific forensic tools, such as EnCase and Forensic Toolkit (S. L. Garfinkel & Shelat, 2003). As deleted files can sometimes be recovered there are tools available for secure deletion.

### Table 1 - Example of false positive rates

<table>
<thead>
<tr>
<th>m/n</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.0138</td>
<td>0.0024</td>
<td>0.0009</td>
<td>0.0006</td>
<td>0.0005</td>
<td>0.0007</td>
</tr>
<tr>
<td>14</td>
<td>0.0177</td>
<td>0.0038</td>
<td>0.0018</td>
<td>0.0013</td>
<td>0.0013</td>
<td>0.0022</td>
</tr>
<tr>
<td>12</td>
<td>0.0226</td>
<td>0.0065</td>
<td>0.0037</td>
<td>0.0032</td>
<td>0.0041</td>
<td>0.0075</td>
</tr>
<tr>
<td>10</td>
<td>0.0329</td>
<td>0.0118</td>
<td>0.0085</td>
<td>0.0065</td>
<td>0.0036</td>
<td>0.0072</td>
</tr>
<tr>
<td>8</td>
<td>0.0490</td>
<td>0.0240</td>
<td>0.0216</td>
<td>0.0255</td>
<td>0.0484</td>
<td>0.0979</td>
</tr>
<tr>
<td>4</td>
<td>0.1549</td>
<td>0.1598</td>
<td>0.2201</td>
<td>0.5128</td>
<td>0.5423</td>
<td>0.7444</td>
</tr>
</tbody>
</table>

Figure 6: Example of Bloom filter false positive rates (Roussev et al., 2006)
which will overwrite the allocated sectors to avoid such discovery, although these tools do not always remove the data completely and some may still remain (Geiger, Matthew, et al, 2006) (S. Garfinkel, 2006). Research on secure file deletion is across operating systems, and there is recognition that while secure file deletion can be a good for benign and legitimate use, it is two-edged and can be used to hide illicit activity, although it is not possible to ever absolutely assure secure file deletion for all time (Bauer & Priyantha, 2001) (Joukov et al., 2006) short of physical annihilation. Sampling of sectors directly off the disk does not use the operating system or file system so can detect hidden files and traces of files which are incomplete. In this way it can defeat anti-forensics techniques which hide the file from the operating system or file system.

2.9 Sampling

It is clearly quicker for data collection and data examination if not all the data needs to be collected and examined to look for a file match. By sampling from a population it is possible to estimate the characteristics of the population as a whole.

Assume a disk of N sectors, containing M sectors of interest. What is the probability of selecting M at random? e.g. N = 30 and M = 4

The probability of not selecting M at random is \( \frac{M-N}{N} = \frac{26}{30} \). The probability of then not selecting M on a second pass is then \( \frac{25}{29} \) and of then not selecting M on a third pass is \( \frac{24}{28} \) and so on. The cumulative probability then of not selecting M in n samples is

\[
\left( \frac{N-M}{N} \right) \times \left( \frac{N-M-1}{N-1} \right) \times \left( \frac{N-M-2}{N-2} \right) \ldots \left( \frac{N-M-n+1}{N-n+1} \right)
\]

This general form of \( \frac{N-(n-1)-M}{N-(n-1)} \) can be written as \( \frac{N-(n-1)-M}{N-(n-1)} \)

The general expression for the probability of not finding M is then

\[
P = \sum_{i=1}^{n} \left( \frac{N-(i-1)-M}{N-(i-1)} \right)
\]

The probability of selecting M is 1 minus the probability of selecting M. In this example, after n samples the probability of not selecting M is, \( \frac{26}{30} \times \frac{25}{29} \times \frac{24}{28} = 0.64 \) and the probability of selecting M is 0.36.

The selecting of random samples without replacement from a mixed population and thus to deduce mathematically something about the population as a whole is the basis for having confidence in survey results (Tuker, 1998). This formula can be used to predict the likelihood of finding a sector of interest when sampling sectors from a disk, and the predicted values can be compared with the actual values experimentally.

2.9.1 Sampling Files on Disk

The random sampling formula does apply, but there is also a simple formula which can be used if files are not fragmented, and there is a regular sampling interval. There does not seem to be much research on this. If files on a disk are contiguous, then given a file of size M sectors, and a regular sampling interval of S sectors then if the sectors on the disk are sampled from a random start point 1-S then the probability, P, of finding the file is;

\[
P = \frac{M}{n}
\]

This is shown by illustration e.g. where M = 6 and n = 8
The sampling points repeat at intervals of 8. In this case the random start point of the sampling is sector 5. There are 8 possible random points where the sampling point can be, 6 of which lie within the file of size M sectors, the remainder lie outside. So of the 8 possible sampling points then 6/8 would find the file. If the file is greater than the sampling interval, then the probability still remains, as each sampling range, within the file will contain a sampling point, and the remainder can be treated as a file smaller than the sampling interval, so the probability is still M/n.

\[ M = 9 \text{ and } n = 8 \]

<table>
<thead>
<tr>
<th>M</th>
<th>n</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>n</th>
<th>M</th>
<th>n</th>
</tr>
</thead>
</table>

In this position there are 2 sampling points within the file. If the sampling point was moved to the right then for 7/8 of the sampling points there would be one sampling point within the file. So the probable number of sampling points within the file is \(2/8 + 7/8 = 9/8\). This simple calculation predicts accurately the results in an Edinburgh Napier Student's Honours Project, Table 16 (Gillespie, 2012) using file size as a correlation with sectors.

<table>
<thead>
<tr>
<th>File size</th>
<th>762kB</th>
<th>762kB</th>
<th>762kB</th>
<th>762kB</th>
<th>762kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling interval</td>
<td>128kB</td>
<td>256kB</td>
<td>512kB</td>
<td>1MB</td>
<td>2MB</td>
</tr>
<tr>
<td>Blocks detected</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Predicted</td>
<td>5.95</td>
<td>2.98</td>
<td>1.49</td>
<td>0.74</td>
<td>0.37</td>
</tr>
</tbody>
</table>

**Table 1: Sampling - predicted hits based on sampling interval and file size**

This is expected for contiguous files and the number of blocks detected must not be less than the leading integer of the predicted value (and will do my own experiments to corroborate). However, files can be fragmented, in which case the probability is more complex and is a problem of finding at least one of several small files. It will still be the case that if the sampling interval is less than the size of the largest fragment then \(P\) must be greater than 1. This is the case of a set sampling interval.

\[ M=9 \text{ and } n=8 \]

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
</table>

As before the probability of the first fragment being found is now \(5/8\), the probability of the second and third fragments being found is each \(2/8\).

The probability of no fragment being found is \(\left(\frac{5}{9}\right) \times \left(\frac{2}{9}\right) \times \left(\frac{2}{9}\right)\)

and so the probability of at least one being found is \(1 - \left(\frac{5}{9}\right) \times \left(\frac{2}{9}\right) \times \left(\frac{2}{9}\right)\)

A related situation where a file has been deleted, then subsequently some sectors have been overwritten. In this case what will remain will be a fragment or fragment less in total than the original file size. For instance, in NTFS and FAT32 when a file is deleted the first letter of the file name in the directory entry is set to 0xE5, but the data remains on the disk and is unaltered (Madison & Dillon, 2006). Those clusters are now available for use, and can be written over by new data, but that could leave fragments of the original data behind, in a form which can still be discovered and identified by sampling sectors. The estimated probabilities can be compared experimentally with the actual rates, but this is only feasible where the fragmentation of the file is known. This knowledge can be used in the lab with a manually crafted fragmentation, but will not be simple to apply in practice.
2.9.2 File Fragmentation

Although a file can be stored anywhere in available space on a disk, it is not usual for the sectors to be scattered. File systems endeavour to write files contiguously to speed up access. Where this is not possible the file becomes fragmented, but it will be fragmented into chunks known as extents or fragments, it will not be atomised into scattered sectors. Even relatively newer operating systems like Windows OS using NTFS have no inbuilt defragmentation, and can only write files contiguously when there is a readily available continuous free space (Danseglio, 2006). However, some operating systems, like GNU/Linux rarely suffer from file fragmentation (Sobell, 2003).

<table>
<thead>
<tr>
<th>File Systems</th>
<th>FAT³</th>
<th>NTFS</th>
<th>UFS</th>
</tr>
</thead>
<tbody>
<tr>
<td># File Systems</td>
<td>219</td>
<td>51</td>
<td>5</td>
</tr>
<tr>
<td># Fragments</td>
<td>Number of Files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Contiguous)</td>
<td>1,286,459</td>
<td>521,663</td>
<td>70,222</td>
</tr>
<tr>
<td>2</td>
<td>25,554</td>
<td>22,984</td>
<td>10,922</td>
</tr>
<tr>
<td>3</td>
<td>4932</td>
<td>6474</td>
<td>1047</td>
</tr>
<tr>
<td>4</td>
<td>2473</td>
<td>3653</td>
<td>408</td>
</tr>
<tr>
<td>5–10</td>
<td>4940</td>
<td>13,139</td>
<td>658</td>
</tr>
<tr>
<td>11–20</td>
<td>1593</td>
<td>7980</td>
<td>94</td>
</tr>
<tr>
<td>21–100</td>
<td>1246</td>
<td>11,901</td>
<td>13</td>
</tr>
<tr>
<td>101–1000</td>
<td>186</td>
<td>5953</td>
<td>0</td>
</tr>
<tr>
<td>1001–</td>
<td>2</td>
<td>590</td>
<td>0</td>
</tr>
<tr>
<td>Total Files</td>
<td>1,326,385</td>
<td>594,237</td>
<td>83,374</td>
</tr>
</tbody>
</table>

Note: this table omits the eight files found on the single UFS2 file system in the corpus (drive 620) and the 16 files found on the single EXT3 file system (drive 1041). The table also omits empty files 0 bytes in length, since they have zero fragments.
³ Includes FAT12, FAT16 and FAT32.

Figure 7: File fragmentation study (S. L. Garfinkel, 2007)

File systems aim to write contiguous files as access is quicker, but will support fragmented files being written. In a survey of 300 second-hand systems bought between 1998 and 2006 in several countries, 6% of files were fragmented (S. L. Garfinkel, 2007). It was considered that fragmentation was higher for files likely to be of forensic interest, so for example 16% of jpeg files were fragmented. The distribution of number of fragments is shown below. This paper was concerned with carving fragmented files and so was interested in the gap between fragments. It was not concerned with sampling to find files which were fragmented, and did not address the issue of how fragmentation might affects sampling interval when looking for a file. The paper does provide real world statistics on fragmentation which show that while fragmentation is not very common it is common enough that the simple formula for the probability of finding a file based on it being contiguous cannot be assumed.

If it was the case that a disk was being sampled which contained several files, such as jpeg, each of which were in the reference set being compared against, and if for simplicity they were similar sizes, then the simple formula for the probability of finding a file based on it being contiguous would tend towards being true. For instance, if 10 files were on the disk, and the probability that any file was fragmented was 0.16, then the probability that all 10 were fragmented would be 0.16¹⁰ which is 0.0000000011, and the probability that at least one file was contiguous is 0.9999999989 and the simple formula would tend to apply. Study of file fragmentation by file type and file size would be a useful addition towards having confidence in selecting appropriate sampling intervals. Combining techniques of sampling and locating files with hashes can support a triage approach.
2.10 **Triage**

Combining the techniques of sampling, hash analysis and matching using Bloom filters which have been described above it is possible to conceive a tool which can look for a file quicker than would be the case with full disk collection and examination. The concept of rationally rationing limited resources by prioritizing needs is well known, and triage is best known for its medical application in life threatening situations. Triage is from the French word 'trier', meaning to sort, triage originally referred to sorting agricultural products but is now most know for its use in emergency medical diagnosis. Where there are insufficient resources to treat everyone, then casualties are sorted and prioritised with treatment targeted at those who will benefit most, and not treating those who are beyond help or who do not need it urgently (Iserson & Moskop, 2007). Its earliest medical application was on the battlefield, and progressively it was adopted, and by the Korean War led to a 'striking' improvement in survival (Jenkins et al., 2008). It is of value where resources are insufficient to treat all casualties, and a method is needed to determine whom to treat.

2.10.1 **Triage in Digital Forensics**

The parallel situation, of limited resources, can arise in computer forensics, where there is insufficient resource in the time available to comprehensively analyse seized or suspect data, and what is required is a technique to quickly identify suspicious data warranting further investigation, while eliminating data which is not suspicious. The role of triage in the digital forensics process has been clearly identified as contributing to closing the gap between growing demand for forensic examination and the limits on investigation resource (Cantrell, Dampier, Dandass, et al, 2012) (Overill & Silomon, n.d.). This triage approach has been modelled for computer forensics in the field, based on an informal but effective approach being used by investigators (Rogers et al., 2006) and there are commercially available tools to support a triage approach such as STRIKE (System for TRIaging Key Evidence) (S. L. Garfinkel, 2010). It has also been applied to mobile devices (Walls, Learned-Miller, et al, n.d.) (Mislan, Casey, & Kessler, 2010). Another triage based approach to digital forensics categorises seized data according to a quick profile of the user (Grillo, Lentini, et al, 2009), this is an interesting approach, but in this study we are ignoring the user and only considering the data.

2.11 **Taxonomy for Digital Forensics Triage**

Taxonomies have been described for digital forensics based on traditional forensics. There seems however to be little research on a specific taxonomy for digital forensics triage. A basic model of 'Acquire, Authenticate and Analyse' the data can be used as a starting point (Sansurooah, 2006) (Pollitt, 2008). While these have been in relation to digital forensics, they have not been specific about triage in digital forensics. A model process for triage in digital forensics can be described which follows the same steps of acquisition and analysis (Rogers et al., 2006). This latter model described triage as a fundamental phase following planning in which items of interest are ranked by priority. Encouraged by this commonality of process models, thought is given here to a model for taxonomy of digital forensics triage which is parameter based and accommodates possible scenarios in which digital forensics triage could be useful.

2.11.1 **Scenarios**

Scenarios where triage has application are varied, and similarly the value of triage can vary. For instance, a single PC might be seized by law enforcement during an investigation and a period of days be available for analysis, compared with customs at a border crossing who may have minutes to carry out routine checks. While the asset might even be similar in these scenarios, such as a laptop, the time available for analysis can be very different.
Other scenarios can arise involving multiple suspect devices, where it is impractical to isolate and analyse all the suspect devices. For instance, if a company is investigating the leak of an internal document it may want to check network shares, and PC hard drives for copies of the leaked file. The company might be in the position of having a suspect pool of PCs, which might even be the entire network. It would clearly be impractical to isolate and examine at leisure all PCs on the network. Similarly, if law enforcement are investigating a corporate email system for threatening emails, it might be impossible to isolate originals or copies of all emails on the corporate email server without stopping the company functioning. The differentiators determining how important triage would be, might then be the size of data to be analysed, and the time available for analysis.

2.11.2 Data Size

The purpose of triage is to prioritise so that limited resources can be used to maximum effect, typically to speed up the analysis of data. Speed of analysis is a function of processing power, time and data. By applying triage the data to be analysed can be prioritised or sampled so as to reduce the actual amount of data to be processed while still achieving a representative result. The problem, and the need for speed, increases with the size of media to be analysed, and for multiple devices it also increases with the number of devices.

2.11.3 Time Window

Another dimension to consider is the time window available, which is affected by the system state and the scenario. Where a system is live and cannot easily be isolated such as a company's email server, or a virtual server running in Amazon Web Services (AWS) then there is more need to act quickly, than a system which can easily be isolated or copied for analysis such as a PC. There is also the constraint of the situation, such as a routine border control check must be completed without undue delay to the traveller, whereas a device seized by law enforcement can be analysed more leisurely.

2.11.4 Taxonomy Based on Speed

<table>
<thead>
<tr>
<th>Data size</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>Border controls</td>
<td>Live corporate server</td>
<td>Significant remote/virtualised system</td>
</tr>
<tr>
<td>Medium</td>
<td>Live corporate PC scan</td>
<td>Live corporate segment scan</td>
<td>PCs network scan</td>
</tr>
<tr>
<td>Large</td>
<td>PC seized in investigation</td>
<td>Server seized in investigation</td>
<td>System imaged for investigation</td>
</tr>
</tbody>
</table>

Table 2: Digital forensics triage taxonomy

This table takes as axes the parameters identified of data size to be analysed, and the time window available for analysis. Suggested example scenarios are shown in the table, but this is not exhaustive. The table is coded to show the importance of using triage to accelerate analysis. Red/Dark Grey – most important, Orange/Medium Grey – important, Green/Light Grey – least important. For testing this triage approach, standard digital corpora can be used.

2.12 Corpora

This triage tool is the basis for a proof of concept design which will need to be tested by experiment. For the experiment results to be repeatable the data sets examined will be based on publicly available models. This is idea behind the common use in genetics of the fruit fly
It is useful to have commonly used standard corpora (data images, including disk images) used for digital forensic research if results are to be comparable (Woods et al., 2011), and such corpora are available (S. Garfinkel, Farrell, et al, 2009). It is obviously problematic to use corpora based on real cases due to privacy and legal reasons, but sample corpora can be explicitly created which use realistic, but not real, evidence. The essence of corpora is standardization in order to support repeatability of experimentation by researchers. Using a publicly available image, and the jpeg files on it, means that the results obtained in experiment can be reproduced and compared by others using the same source material. As well as using corpora, this experiment can be supported by using a cloud based environment.

2.13 Cloud-based Virtualized Environments

A suitable environment for carrying out the experiments is required. The limitations on local PC capacity presented a threat to running experiments to process large amounts of data, while the availability of a computing cloud at Edinburgh Napier University presented an opportunity to run a virtual lab unhindered by local limitations. Cloud-based virtualised environments present the user with an interface to access computing resource, which hosted in software can mimic hardware configurations, such as a server. In a commercial environment it can offer several advantages such as extensible computing power on demand, on a pay-as-you-go basis, rather than an up-front investment with maintenance overheads (Armbrust et al., 2009). It also has benefits in academia (Sultan, 2010). The cloud can break the student's reliance on local PC performance and availability, but creating a virtual resource which can be accessed from home or university. The Edinburgh Napier University cloud will be used for this study for that reason, although the issue of resource contention in the cloud has been recognised, the Napier cloud is proving effective in teaching (Buchanan, Graves, et al., 2011) and in research into a Digital Forensics Evaluation Test (D-FET) platform. (Buchanan, Macfarlane, et al., 2011) (Buchanan & Macfarlane, 2011). It has been reported that while there has been much research on putting applications in the cloud, there has been less research on testing them (King & Ganti, 2010).

2.14 Conclusion

This chapter has researched several topics relevant to the aims and objectives of the thesis. There is real world application for a technique which enables faster analysis of data, such as in searching for a particular file. When it comes to crime, the digital world has increased the capabilities of the criminal and the crime fighter (Nuth, 2008). There are several situations where it can be necessary to examine a data storage medium to look for a particular file, or file, where the file directory cannot be used to do a simple file name lookup, but instead the medium needs to be directly examined to see what content it contains.

One situation would be where a file is lost or misplaced by the user. That is a benign instance. Less benign is where a medium, such as a computer hard drive has been seized as part of an investigation. This could be a criminal investigation, or an internal organisation investigation, which could be looking for illegal or unauthorised content, such as child pornography images or pirated software. There can also be occasion, such as border controls, where there is value in being able to screen suspect media while constrained by the limited time in which to delay the traveller. In all cases, there is a requirement to scan media for content which is not identifiable by searching the file directory. The requirement is for a fast method of analysing suspect media for content.

The increase in the use of devices with increasing data storage capacity is increasing the burden on forensic investigators, and is beyond what can be met by increased processing and data transfer rates for investigations. The techniques of statistical sampling to search a
population are not new. Neither is the concept of triage which such sampling supports when applied to disk. The concept of hashing is not new either, and logically applies to distinct sectors as well as to whole files. None of these ideas are new or unproven, and in combination they provide a strong theoretical basis for the technique of sampling sectors on a disk to look for a particular file. That technique can be supported as a process by the use of Bloom filters which allow the rapid matching of sampled values against a large database of reference values, with a predictable level of accuracy. There seems to be little research on a taxonomy for digital forensics investigation triage, and that is developed a little here based on the application of triage. Using standard digital corpora in the experiments allows repeatability and for others to validate, or challenge the results obtained here. Experimentation can also be supported by using a cloud based experiment environment.
3 Design

3.1 Introduction

This chapter describes the design for programs and experiments to test the following specific hypotheses based on the aims and objectives and based on the techniques identified in the research in the Literature Review chapter.

1. A prototype digital forensics investigation tool can be created using, sampling, sector hash analysis and a Bloom filter.
2. Using a Bloom filter will give a faster match than a database lookup and can have an acceptable false positive rate.
3. The constraint on speed of finding a file on the target is the speed of matching against the Bloom filter, rather than the speed of reading the sectors and hashing them.
4. For contiguous or fragmented files a regularly spaced sampling interval less than the file size or largest fragment size guarantees the file will be located. This can be tested and confirmed.

The design was for programs which could make a database and a Bloom filter bit array by hashing the sectors of a selected file. Tools would then be designed to read sectors from a disk image on file, generate hash values and match them to the file using the database and Bloom filter. A set of experiments would then be designed to test the effect of changing Bloom filter parameters on speed and accuracy of locating the file. Experiments would also test the effect of file fragmentation.

3.1.1 SHAFT

The tool is a proof of concept for a digital forensics investigation triage tool using disk sector hash analysis, sampling and Bloom filter or database for matching. For convenience this tool will be given the acronym SHAFT (Samping, Hashing Analysis Forensic Tool).

3.2 Overview

The tool is a proof of concept of a tool that could be used in a variety of situations to look for contraband material. This could be devices or media seized in an investigation or intercepted at a border control which might contain contraband material.

![Figure 8: Design Overview - scenario](image-url)
In such a scenario the requirement is to implement the features described in the literature review, to sample sectors off the suspect media or device, hash these, and compare those hashes with a database or Bloom filter to check for match against a file.

**3.3 Design Methodology**

There are several different software design methodologies. For this design the 'waterfall' method was used. Although it does not allow for much revision, and can be problematic for complex application, for simple applications like this it is effective (El-Haik & Shaout, 2010).

The concept and requirements have already been identified. The design stage is relatively straightforward as the program is linear in that it is simple with few processes and can be described in a single diagram.

**3.4 Program Design**

In order to test the use of sampled sector hashes for locating a file on a disk, several programs are required. First the file which will be searched for needs to be used to generate sector hashes. These are the sector hashes which will later be compared against the sector hashes from sampled disk sectors when looking to see if the file is on a disk. The hashes from the file, one for each distinct sector, need to be stored in both a database and a Bloom filter bit array so that later they can be used to check for a match. With the database and Bloom filter
bit array prepared, the next stage is to have programs which sample sectors from a disk, generate sector hashes, and then check if these match against the file using either the database or Bloom filter to check. When doing experiments it will also be necessary to record the results.

### 3.4.1 Make Database and Bloom Filter

The program for producing the database and Bloom filter needs to first initialise these to zero. The program then sequentially reads every sector of the file. Each sector is hashed and if the hash is distinct it is written to the database and also used to set indices in the Bloom filter bit array. This process is repeated until every sector in the file has been read. The program design can be illustrated as below.

![Figure 11: Program design - Make database and Bloom filter](image)

### 3.4.2 Use Database for Matching Hashes

After creating a database which stores hashes for all the distinct sectors in the file, experiment can then be carried out against a disk, or disk image, to see if the file may be present. This program samples sectors and does not read the entire disk image. Each sector is hashed and that hash is looked up in the database. If there is a match this is displayed. Whether it matches or not the next sector is sampled and hashed. It would be possible to break the program on the first match, as at that stage the file has been located, however that would be artificial in the experiment environment where a standard disk image has been created, and would simply mean the process stops at about the same point in the disk image each time the program runs. The program design can be illustrated as below.
3.4.3 Use Bloom Filter for Matching Hashes

From the literature review it was apparent that Bloom filters could offer speed benefits in matching (Farrell, Garfinkel, et al., 2008). It was also attractive to see if a Bloom filter could be implemented and how it would perform. The program to use the database is still useful because it can be used as a check on the Bloom filter results, and is also insurance against the possibility of the Bloom filter program not working as predicted. After creating a Bloom filter bit array which also 'stores' hashes for all the distinct sectors in the file, experiment can then be carried out against a disk, or disk image, to see if the file may be present. Again, this program samples sectors and does not read the entire disk image. Each sector is hashed and that hash is used to set indices using the same algorithm as used to generate the Bloom filter bit array. If these indices are all set in the Bloom filter then that is a match, although with the Bloom filter this could be a false positive. If there is a match this is displayed. Whether it matches or not the next sector is sampled and hashed. It would be possible to break the program on the first match, as at that stage the file has been located, however this may be a false positive match. The program design can be illustrated as below.
3.4.4 Use Bloom Filter for Matching Hashes and Record Results

A fuller program was designed for use in the experiments. Comparing the overall program performance of the two programs 'Use database' and 'Use Bloom filter' will show whether and to what extent the Bloom filter match is faster than the database. To analyse program performance in detail a second 'Use Bloom filter' program is required. This is similar to the Bloom filter program which is immediately above. It has additional functionality in that as well as displaying results to screen it writes output to a csv file for later analysis. Comparing the overall program performance of the two Bloom filter based programs will show if the metrics themselves are impacting upon performance, and if this is the case it can then be calculated and compensated for in the evaluation. The program design can be illustrated as below.

*Figure 13: Program design - Use Bloom filter*
3.5 Experiment Design

3.5.1 A Scientific Method

The intention is to conduct experiments which test a hypothesis and are reproducible. There has been discussion about the extent to which computer science is a science, that is something which is testable and falsifiable. Back in 1998 Tichy identified a lack of experiment in computer experiment compared to other sciences and hoped to see experimentation in computer science 'flourish' (Tichy, 1998). Denning believes that the 'science paradigm' will be part of mainstream computer science (Denning, 2005) by which he means the method dating back to Francis Bacon of forming a hypothesis and testing it by experimentation. That the
subject of inquiry in computing is usually information, compared with the physical universe in physics, does not affect the applicability of this scientific method (Dodig-Crnkovic, 2002). Mocas identifies that reproducibility is part of the scientific method, but is equivocal about whether digital forensic evidence gathering is always reproducible, even though that is preferable (Mocas, 2004). The use of standardized digital corpora is one element which can help provide reproducibility in testing and experimentation (S. Garfinkel et al., 2009).

The experiments will be documented and described adequately so that similar experiments could be repeated by others. The intention is for experiment to be methodical and scientific. Standards for general software verification and validation have been around for some time (IEEE, 1998). There is research specifically on testing methodology for digital forensics tools, which is what the sample sector hash matching program is. A valid method is to define the functional requirements of the tool and how that functionality can be evaluated, then to use standard reference sets, such as corpora to build test cases, then to carry out the experiments and generate, record, analyse and evaluate metrics (Beckett & Slay, 2007), and more specific research on methodologies for testing specific functions has been carried out by others, for example for copying (Guo & Slay, 2010) and for searching (Guo, Slay, et al, 2009). Others have developed a common evaluation scheme for forensics software (Hildebrandt, Kiltz, et al, 2011).

![Diagram of program algorithm](image)

Figure 15: General experiment program flow

### 3.5.2 Base Lining Experiments

Baseline experiment will first establish the effect of what might be termed environment factors. These factors would be include how the program is executed and the environment it is run in. The data source type would also be a factor to baseline. As well as establishing benchmark times, the base lining will quantify the effect of these environmental factors to identify which might be significant so that later when the parameters for Bloom filter settings are varied then any change in output such as program times can more reasonably be attributed to the change in the parameters. The experiments were to be repeated, both repeated as a set number of cycles when the experiment program is run, and also for the program to be run regularly on a scheduled basis using the crontab function. This is illustrated in the figure above. The purpose of this repetition is to provide more data to provide a smoothing effect with any extremes to be diluted by the other experiment results. This could eliminate the effect for example if a particular experiment just happened to coincide with a background process, or contention for resources within the remote VM environment.
3.5.3  Bloom Filter Experiments

Once base lining has been established then the Bloom filter parameters will be varied to see what effect that has on outputs. The measured outputs will be program times, hit rate (the proportion of experiments which have a positive hit on the search file) and accuracy (an indication of how many of the positives are true positives and how many are false positives). The two variable Bloom filter parameters are the bit length of the Bloom filter \( m \) and the number of hashing functions \( k \) used to set the Bloom filter. Other variables for experimentation are the source file size and the sampling intervals. Different permutations of these parameters would be tested. These are the only parameters a Bloom filter can take, and it was important to assess if the false positive rates were as predicted, and also to see what the effect on speed is of varying these.

3.5.4  Fragmentation Experiments

Similar to Bloom filter settings, once base lining has been established then experiment will be carried out to see the effect of fragmentation. Fragmentation will require images with an unfragmented file to be directly manipulated to create copies with the file in an known number of fragments, each of a known size and location within the image. In practice the fragmentation of files on disk is usually not known. The aim of the experiments was to create artificially fragmented files to assess if the hit rates were as predicted and to measure if there was any effect on speed or accuracy.

3.5.5  Metrics

The metrics to be recorded for later analysis will be:

Inputs to the experiments:- the data source size and media, the interval between sampled sectors off the disk, the program run environment (how and where the program is run), the extent of fragmentation in the data source, the Bloom filter settings for the array bit length and the number of hashing algorithms, and the number of times the experiment is looped and run by the program.

Outputs from the experiments:- the overall time the program takes to complete, disk time - the time the disk read and hashing within the program takes, match time - the time the matching within the program takes, the number of matches made, and whether a match is a true positive or a false positive. For any eventual practical triage tool then speed, hit rate and accuracy are the essential measures, and the proof of concept prototype tool would need to identify what might effect speed, hit rate and accuracy.

3.6  Conclusion

This chapter has provided program designs to create a digital forensics investigation triage tool and experiment design to test the hypotheses based on the aims and objectives and based on the techniques identified in the research in the Literature Review chapter. A scenario was outlined and from that a basic process was outlined for the tool. A program design was produced to support the creation of the database and Bloom filter bit array to store the sector hashes of the file. Program designs were also produced to create a tool which could use the database or the Bloom filter to test if the file could be located on a disk image. A series of experiments were designed on a scientific basis to investigate the effect on speed and accuracy of file location of varying Bloom filter parameters, and file fragmentation. These experiments were to be repeated multiple times over a period to give a reliable measure.
4 Implementation

4.1 Introduction

The design of the programs and procedure to be used has been presented in the previous chapter. This chapter describes the working implementation of the experiment programs and experiment environment. The design requires four programs to be written, namely programs to create the database and Bloom filter bit array, to use the database for matching, to use the Bloom filter for matching, and to use the Bloom filter for matching while taking measurements of performance. As well as these programs, the scientific method has been described and provides the basis for the experiments to be implemented to investigate the effects of varying Bloom filter parameters and of file fragmentation.

4.2 Objective

The objective is to produce a functioning tool for sector sampling and hashing, and identifying matches with the file being sought – which will always be the same, single jpeg file and against a 256MB and 2GB image containing an unfragmented file. First, baseline performance will be established. The baseline experiments will establish the effect of environment factors, such as how the program is run, if it is run on the local PC or remote PC, if performance is affected by time of day, if performance reading a disk file image is different from using an attached UBS flash drive. Experiments will then examine the effect of changing the parameters for Bloom filter settings for m (Bit size) and k(number of hash functions). Accuracy and speed, the effect of fragmentation across a range of data sources and sampling intervals.

In each case the experiments will be run several times and results given as average, minimum and maximum for times, hits, and false positives. For consistency the experiments will be run as much as possible on the PC while no other applications are running. The computing cloud environment is uncontrolled by the local user, but repeating experiments and comparing resulting times can indicate a baseline and may identify variations in the cloud performance.

4.3 Images

In this case the requirement can be kept simple, as a search for one particular files amongst a collection of other files. The image nps-2009-canon2 (30MB FAT16 SD card image) from the Digital Corpora website will be used (Digital Corpora, 2012). This contains 36 jpeg files. From this IMG_0036.JPG was chosen to be the file searched for, and the other 35 jpeg files would be used for contrast. The file IMG_0036.JPG is 882 337 bytes in size and occupies 1724 sectors of 512bytes each on disk. Larger corpora are available, for example the Digital Corpora website has a download of over 100 000 images, so the corpora being used in these experiments is a basic one.

As well as the 30MB disk image file which was used for development, 256MB and 2GB USB memory stick each were formatted and sectors cleared using dd. Then the memory sticks were filled with multiple copies of the 35 jpeg files, and a single copy of IMG_0036.JPG. These memory sticks were then imaged using dd and it was confirmed that the file IMG_0036.JPG was contiguous. The dd command was then used to make copies of the memory sticks in which the file IMG_0036.JPG was fragmented manually
a) into 2 fragments, one of 1146 sectors and the other 575 sectors
b) into 3 fragments, one of 574 sectors and two of 575 sectors

The larger 256MB and 2GB images (6 in total) were then used in the labs for experiment. The images are described in detail in Appendix 10.1 and illustrated in overview below.
It is possible on a disk that files be hidden in a HPA (Host Protected Area). The optional HPA is designed for disk vendors to store disk configuration data, and it not used by the file system to store files. If it exists it can be detected and the sectors examined, but this requires active investigation, and changing the disk configuration (Carrier, 2005). In this case it is assumed files will not be in the HPA, and the search will be limited to the sectors of the disk shown in the file system. For simplicity single sectors were used for hashing and sampling rather than small blocks of sectors.

### 4.4 Bloom Filter Implementation

As has already been described, it is inherent with Bloom filters that they can give false positives as a result of collisions. This is where a value which is not in the reference set is hashed and matches with the Bloom filter because it overlaps with a composite of matches for several values which are in the reference set. These collisions are inevitable, although the rate depends on the Bloom filter settings.

Particular to sector hash analysis there is however another false positive which is avoidable. The concept of distinct sectors has already been described, that is that a sector is distinct if it is unique to a file and therefore its presence is a guarantee that the file it came from, and no other, is present. The possible combination of values at random for a sector is $512 \times 2^8$, an astronomical number. Of course sectors are not filled at random, and some sectors are 'indistinct' that is they appear in more than one file. It then follows that the detection of an indistinct sector is not a guarantee that the file it came from, and no other is present. The clearest, and commonest indistinct sector is 'all nulls'. This is common where nulls are used to stuff unused sectors, the 'slack space', in a cluster, and this slack space can be used covertly to conceal data (Berghel, Hoelzer, et al, 2006), but in normal use will be stuffed with nulls.

For example, a file may be 15 sectors in size on a file system organised into clusters each of 8 sectors. The 15 sector file requires 2 such clusters, and the unused 16th sector may be stuffed with nulls. For the experiments the jpeg file IMG_0036.JPG from the (30MB corpora ID here) was chosen to be the 'bad file' being searched. Before creating a Bloom filter of its sector hashes it was necessary to identify any indistinct sectors and then avoid using them when creating the Bloom filter. This was done by hashing all sectors on the 30MB corpora ID.
and then using the counter module within Python to identify any duplicate, and hence indistinct sectors. The duplicate sectors within the IMG_0036.JPG were thus identified, and as expected this included the sector of all nulls. There is of course the possibility that sectors would remain which might be found on other media in other files, but that is unavoidable.

Experimentally it would still be possible to detect any such indistinct sector matches later by noting when a positive match is found whether the match occurred on a sector within the known sector allocation for the bad file. False positives would be identified in a 2 step approach. When the Bloom filter matched a sector which was outwith the known sector allocation, that sector hash could then be looked up in the database of all the bad file sector hashes. If the hash value was not in there, then it was a false-positive caused by a Bloom filter collision. If the hash value was in the database then as in all cases the allocated sectors for the bad file would be known it was therefore a sector from another file but identical to a sector in the bad file, and was thus a false positive due to being an indistinct sector. Of the 1725 sectors in the bad file IMG_0036.JPG, 1718 were distinct, and 7 were indistinct.

A decision had to be made about the parameters to be used in the Bloom filter. It was convenient to generate hash functions for the Bloom filter indices by using bytes from the MD5 hash of the sector. The MD5 hash is 32 bytes (128 bits), which conveniently could generate 8 x 4 byte chunks (8 x 16 bits), each being 4 bytes was adequately distinct and each chunk mapping to one of \(2^{16}\) possible values, which would map directly to a Bloom filter with a bit length of \(2^{16}\).

The predicted false positive rate for a Bloom filter has already been described and is

\[
P_{fp} = (1 - e^{-\frac{kn}{m}})^k
\]

In this case the values will be:

| n = 1718 | This is the number of distinct sectors in the bad file IMG_0036.JPG, each of which will be hashed and used to set indices in the Bloom filter. |
| m = \(2^{16}\) = 65536 | This is the bit length of the Bloom filter. |
| K = 8 | This is the number of hash functions applied to the MD5 sector hash in order to set an index in the Bloom filter bit array. |

This gives a value of \(P_{fp} = 0.00000164\)

For convenience \(P_{fp}\) was calculated for \(m = 2^{12}\) also, as that could be mapped to chunks of 3 bytes from the MD5 hash of the sector. For both \(m = 2^{16}\) and \(m = 2^{12}\) , values of \(k = 8\) and \(k = 4\) were used for comparison. A tabulation of the results for \(P_{fp}\) is:

<table>
<thead>
<tr>
<th>M (BF bit length)</th>
<th>K (# of hash functions)</th>
<th>Predicted FP rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536 ( (2^{16}) )</td>
<td>8</td>
<td>0.00000164</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.00009820</td>
</tr>
<tr>
<td>4096 ( (2^{12}) )</td>
<td>8</td>
<td>0.75266841</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.43731715</td>
</tr>
</tbody>
</table>

Because of the advanced maths, and the complexity of understanding how the Bloom filter probabilities compute, at an early stage it was decided to run some experiments to assess whether the program written to compute matches using Bloom filters agrees with the predicted false positives. Initially the experiment for false positive was run against the 30MB USB memory stick, containing the bad file 'IMG_0036.JPG' and 35 other good files. This was done sampling every 100th sector, and with 100% of all sectors.
Initial evaluation of Bloom filter parameters, predicted false positive vs actual (10 iterations)

<table>
<thead>
<tr>
<th>M (BF bit length)</th>
<th>K</th>
<th>Sectors sampled</th>
<th>Predicted FP rate</th>
<th>Predicted FP count</th>
<th>Actual count</th>
<th>FP Actual/Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536 (2^16)</td>
<td>8</td>
<td>6080</td>
<td>0.00000164</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6080</td>
<td>0.00009820</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>4096 (2^12)</td>
<td>8</td>
<td>6080</td>
<td>0.75266841</td>
<td>4576</td>
<td>4468</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6080</td>
<td>0.43731715</td>
<td>2659</td>
<td>2481</td>
<td>0.93</td>
</tr>
<tr>
<td>65536 (2^16)</td>
<td>8</td>
<td>60799</td>
<td>0.00000164</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>60799</td>
<td>0.00009820</td>
<td>6</td>
<td>10</td>
<td>1.67</td>
</tr>
<tr>
<td>4096 (2^12)</td>
<td>8</td>
<td>60799</td>
<td>0.75266841</td>
<td>4576</td>
<td>44445</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>60799</td>
<td>0.43731715</td>
<td>26588</td>
<td>24833</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Constants: n = 1718; image file = 30Mnpsf1.raw; target file = img_0036.jpg; K = # of hash functions

Table 3: Bloom filter false positives - predicted vs actual (30MB)

The results were encouraging, but not exact. The reasons for the discrepancy could be due to the particular distribution of sectors on this disk, but the high correlation gives confidence that the program is implementing the Bloom filter correctly, and is consistent with the false positive rate predicted by theory. It was decided to repeat the experiments on a larger scale, against a 256MB USB stick. First with a stick containing a single copy of the bad file, and with the 25 good files copied hundreds of times to fill the stick.

<table>
<thead>
<tr>
<th>M (BF bit length)</th>
<th>K</th>
<th>Sectors sampled</th>
<th>Predicted FP rate</th>
<th>Predicted FP count</th>
<th>Actual count</th>
<th>FP Actual/Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536 (2^16)</td>
<td>8</td>
<td>489471</td>
<td>0.00000164</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>489471</td>
<td>0.00009820</td>
<td>48</td>
<td>85</td>
<td>1.77</td>
</tr>
<tr>
<td>4096 (2^12)</td>
<td>8</td>
<td>489471</td>
<td>0.75266841</td>
<td>368409</td>
<td>361720</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>489471</td>
<td>0.43731715</td>
<td>214054</td>
<td>213984</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Image file = 256Mjcusbf1.dd

Table 4: Bloom filter false positives - predicted vs actual (256MB)

For the 2^16 bit length the actual and predicted totals are minuscule and consistent. Though the numbers are small the actual false positives for the 2^12 bit length Bloom filter are very slightly less than predicted. Interestingly this is in contrast to a paper (Bose et al., 2008) which contends that the established formula for Bloom filter false positives will underestimate the actual false positive rate, though by a slight amount which can be negligible for a large value of M and a small value of K.

4.5 Program Implementation

A decision was made to use the SleuthKit and Python in the implementation. Python is a programming language, selected in this case as a new language to learn, and also because of its use with the SleuthKit. The SleuthKit is a known set of command line tools and library which can be used for digital forensics there is also a graphical user interface (GUI), Autopsy (Anon, n.d.-b). Mostly developed by Brian Carrier, it can read raw image files and extract meta data and files. Garfinkel in 2009 described the SleuthKit interface as not well suited for to being called by programming languages, and he wrote a program, fiwalk, to produce xml which could more easily be ready by Python. (S. L. Garfinkel, 2009). Garfinkel has done further work with this as he developed DFXML, and XML schema for digital forensics. (S. Garfinkel, 2012) There is a Python binding, pytsk to make the SleuthKit library available and this could be used.(“pytsk - Python bindings for the Sleuthkit,” n.d.), however while the
binding was tested successfully later, in the event this functionality was not incorporated or used in the programs. Others have used Python for digital forensics tools, the Volatools digital forensics toolkit is implemented in Python (Walters & Petroni, 2007), as is PyFlag used for network forensics (Cohen, 2008) and a taxonomy of Python libraries useful for forensics has been produced (O’Connor, 2010). Python is extensible and to support the implementation of the Bloom filter, the Python bit array module was added, as this is not included in the standard Python distribution.

An early decision to use Python 2.6 instead of Python 3.0 was precipitated by the preponderance of 2.6/2.7 code on the internet. All the functionality of 2.6 could be delivered with 3.0, and 3.0 could be used in future, but it was easier learning 2.6. Python was chosen to learn because of its perceived popularity and relative simplicity. It is not best known for performance. The experiments will analyse the speed taken by the program, and it is possible that a future comparison could use code written in a different language. The default development environment supplied with Python is IDLE, which supports syntax handling and proved adequate for development.

There were 4 main programs written in Python (identifiable with the py extension) for experiment.

- **Makebf.py** – This program would create the Bloom filter, and also create a database of sector hashes. All the distinct sectors in the reference bad file IMG_0036.JPG were read and hashed with MD5. The sector number and MD5 hash value were added to the database. The MD5 hash was broken down and used to set indices in the Bloom filter.

- **Usedb.py** – This program would read sectors on the disk and lookup for a match in the database. Provided all the sectors were distinct there would be no false positives.

- **Usebf.py** – This program would read sectors on the disk and lookup for a match using
the Bloom filter. Provided all the sectors were distinct there could still be false positives.

- Usebmfmetrics.py - This program would read sectors on the disk and lookup for a match using the Bloom filter. In addition it took several metrics, such as timing sections of the program, and it would use the database to check whether any positive matches in the Bloom filter were true or false, and if false whether it was a Bloom filter collision or an indistinct sector. A log is written to results.txt and data is written to results.csv.

All these programs could take variable parameters, and the 'Use' programs could loop to repeat experiments for a number of cycles, and to loop through variable parameters against a constant algorithm. The design for the programs is described in section 3.3, and the full source code for the programs is listed in Appendix 9.2.

### 4.5.1 Make Database and Bloom Filter

The program to create the database and Bloom filter from the sector hashes from the search file was named makebf.py. This program initialises various parameters, including a list of hashes of indistinct sectors which are common to sectors on other jpeg files in the corpora (including all the nulls sector)

```python
# md5sum for indistinct sectors obtained by experiment on raw image
dup = ['bf6f19ea0c3df36f84b94e9344137e8b', '890c1bafla03033bf8841d956f24961f',
       '0329e8ac3f4c4db486d4b89f6ea56c', '902600c5d0b1268b8a3d9328705be438',
       'f16bb3da25ed4db7f8205e724cbfb', 'f030cbbdc28d1979156d9da24049436',
       '2de01f616a64d1795d21a2715c1cfa', '897b52a357c01337344749a304407388',
       '3858ea3b4f148747f1691f02f9210c90f', '19f8b4bc9505a56627b60d3177d3f7e8',
       'a4ecdb7f9503d5d453e8c590']
```

The database is initialised then populated with the hashes for each sector in the source file (IMG_0036.JPG). The hashes are generated by using the Unix dd command and md5sum operating system commands.

```bash
c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
o.s.system(c)
result=os.popen(c).read()
if result[32] not in dup:
distinct = distinct + 1
d = "INSERT INTO Hashes VALUES('%s','%s','%s')" % (s, result[32],
```

This produces the database entry for the sector hash. The same hash value is then used to set indices in the Bloom Filter bit array. The number of hash functions used to set the Bloom filter bit array is a variable k (this should not be confused with the hash function used to generate the md5 hash value for the sector). The md5sum value is a 32 character hexadecimal. Where for example the Bloom Filter bit length is \(2^{16}\) bits, then a 4 character hexadecimal gives \(2^4\) or \(2^{16}\) possible values. So a sequence of 4 character hexadecimal values from the md5sum hash can be extracted, each one then converted to set to 1 an index in the Bloom filter. This is repeated for consecutive hexadecimal values k times. The result is that the Bloom filter bit array is set to 1 in k locations, each with an index value between 0 and \(2^{16} - 1\).

```python
for i in range(k):
    idx = int(result[i*4:(i*4)+4], 16)
    with open(bloomfile, 'wb') as fh:
        fh.write(bf.tofile(fh))
```

This is repeated for every sector in the jpeg file, to give a database of hashes for each sector, and a Bloom filter bit array with some indexes set to 1.
4.5.2 Use Database for Matching Hashes

The program which used the hash database to check for matches in the disk image was named usedb.py. This program in a similar way to the makebf.py program uses the OS commands dd and md5sum to read sectors and generate hashes. The difference is that instead of every sector being read and hashed, a sample are read and hashed. The sampling interval sint is a variable, and starting at a random sector between 1 and sint, then sectors are read spaced apart by the sampling interval.

```python
for s in range(sstart,sstop,sint):
    match = 0
    c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
    os.system(c)
    result=os.popen(c).read()
```

The database of hashes is then used to see if there is a match on the hash value this is displayed

```python
d="Select * FROM Hashes WHERE Hash='%s'" % (result[:32])
cur = con.cursor()
cur.execute(d)
data=cur.fetchone()
if data:
    print str(j), "db Sector ", s , " hash ", result[:32], " is a match"
```
Use Bloom Filter for Matching Hashes

The program which used the Bloom filter to check for matches in the disk image was named useby.py. In this program sampled sector hashes are generated in the same way as in the usbdb.py i.e.

```python
for s in range(sstart,sstop,sint):
    match = 0
    c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
    os.system(c)
    result=os.popen(c).read()
    for i in range(k):
        #print[int(result[i:i+4], 16) % 2**16] , bf[int(result[i:i+4], 16) % 2**16]
        if bf[int(result[i*b:(i*b)+b], 16)] == 0:
            #if bf[int(result[i:i+b], 16)] == 0:
                match = 0
            break
        else:
            match = 1
    if match == 1:
        print str(j) , 'bf Sector ' , s , ' hash ' , result[:32] , ' is a match.'
```

The difference in the two programs is of course the comparison, which is using the Bloom filter bit array. For the md5sum hash value, the same algorithm is used as was used in the makebf.py program. If for the hash of a particular sector all k hash Bloom filter hash functions, there is a match in the Bloom filter bit array then that is a Bloom filter match and is displayed.

```python
for i in range(k):
    #print[int(result[i:i+4], 16) % 2**16] , bf[int(result[i:i+4], 16) % 2**16]
    if bf[int(result[i*b:(i*b)+b], 16)] == 0:
        #if bf[int(result[i:i+b], 16)] == 0:
            match = 0
        break
    else:
        match = 1
    if match == 1:
        print str(j) , 'bf Sector ' , s , ' hash ' , result[:32] , ' is a match.'
```
4.5.4 Use Bloom Filter for Matching Hashes and Record Results

The program which used the Bloom filter to check for matches in the disk image and also recorded metrics from the experiments was named usebfmetrics.py. This program is like the useby.py program, but with several additions. It records performance metrics, and is able for instance to determine if a match is a true positive as it includes values for the sectors in each image which are occupied by the search file e.g.

```python
if iimage == "256Mjcusbnewpic.dd":
    vstart0, vstop0, vstart1, vstop1, vstart2, vstop2, sstop =
    57464, 59187, 57464, 59187, 57464, 59187, 489471
```

and where a false positive is detected it looks up in the database to determine if the hash is in the database, meaning the hash is for an indistinct sector which is present in another file, or else that it is a Bloom filter false positive.

```python
if match == 1:
    if (vstart0 <= s <= vstop0) or (vstart1 <= s <= vstop1) or (vstart2 <= s <= vstop2):
        tpos += 1
        print j, ' Sector ', s, ' hash ', result[:32], ' is a true +ve match'
    else:
        con = lite.connect('hashes.db')
```

Figure 20: Program implementation - Use Bloom filter
The program also writes the metrics to a csv file for later analysis.

```
fcsv = open('results.csv','a',0)
    fcsv.write(str(datetime.now()) +"," + str(datetime.time(datetime.now()))+ "," + comment + "," + str(t) + "," + str(sampled) + "," + str(m) + "," + str(k) + "," + iimage + "," + str(frag) + "," + str(sstop) + "," + str(sint) + "," + str(iimage) + ","
```

A diagram of the program implementation, based on the program specification from the design stage, shows the names of files used, and illustrates the iteration of experiments over a number of cycles.
4.6 Experiment Environment

There were two experiment environments, which were functionally identical in that both would be running the same OS and support the same programming language for experiment purposes. The local PC which would be adequate for initial testing of programs for functionality, with a higher specification setup in a remote virtualised environment which could be used for testing programs more thoroughly. This virtual environment also had the advantage that it could be accessed from a range of locations, provided an internet connection was present and the VMWare web browser plug-in could be installed.

Figure 21: Program implementation - Use Bloom filter with metrics
4.6.1 Local PC

Development and initial experiment was done on a local PC at home. This was accessible and controllable and provided a convenient means of working experiments up from initial conception to first experiments. Data backup was considered essential. The source data could be backed up on memory sticks. The working documents, including the programs, could be backed up in Ubuntu One. Ubuntu One works as a dynamic backup, but is not designed as an archive, that is files are continually saved even if that leads to overwriting an earlier version which is more correct. To allow for this manual backups of key files were also taken and put in an archive folder, and that archive folder was also dynamically backed up in Ubuntu One. Finally the report was saved in Dropbox. This was principally for sharing with the thesis supervisor, but also provided another backup.

4.6.2 Remote Virtual PC

Further experiment was done on a remote virtual PC hosted in the Edinburgh Napier University School of Computing cloud. This was chosen for several reasons. It provided an experiment environment which could be equally accessed from both home and campus. It also provided more computing resource and performance than the local PC. The resource available to the remote PC can be increased and is scalable. The cloud environment itself should be more robust than a local PC. The virtual PC can at a later date be reused, or shared with others, and is not as restrictive. Source images were copied from the local PC to the remote PC using a combination of Dropbox and Ubuntu One. Any data generated on the remote PC was copied via Ubuntu One to the local PC and from there was also backed up as already described for the local PC.

The specification of the local PC and the Remote VM PC is as follows;

<table>
<thead>
<tr>
<th>Local PC</th>
<th>Remote VM (on VMware ESXi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acer Aspire Revo 3600</td>
<td>Single core CPU</td>
</tr>
<tr>
<td>Intel Atom 1.6 GHz single core 512kB L2 cache</td>
<td>4GB RAM</td>
</tr>
<tr>
<td>1GB DDR2 RAM</td>
<td>40GB HD</td>
</tr>
<tr>
<td>160GB HD</td>
<td>Ubuntu 10,04</td>
</tr>
<tr>
<td>Ubuntu 10.04</td>
<td></td>
</tr>
<tr>
<td>USB1</td>
<td></td>
</tr>
</tbody>
</table>

*Table 5: Hardware specification - local PC vs remote PC*

The local PC in particular is relatively low powered, which is a factor in considering the application of timings to any real world application. A comparison with timings on the Remote VM may indicate how significant the hardware specification is, although a truer comparison would be with a higher specification local PC, unhindered by any complexity of resource availability within a virtual environment.

4.6.3 USB Flash Drive

Additional hardware used were a 256MB and 2GB USB flash drive. These were used to provide input for a disk image file. This implementation using a static disk image is a dead box forensics situation and has no aspect of live forensics. The USB flash drives could also be used attached directly for experiment on the local PC. This could not be replicated on the remote virtual PC for experiment.

4.7 Experiment Procedure

The experiment tools were written in Python, and could be run locally on the home PC, or
remotely on the VMware virtual PC (VM). The general process was to test and debug code on the home PC using the 30MB disk image. Once code had been tested and proved, experiments were then run against the 250MB disk image on the local PC and the VM, and against the 2GB disk image on the VM. The output of experiment was summarized in a text file written to results.txt and logged in more detail to a csv file results.csv. These files were cumulative between experiments, and the csv file allowed results to be analysed. These results were tabulated and charted for inclusion in this thesis.

### 4.7.1 Bloom Filter

The Bloom filter variable parameters are bit length ($m$) and number of hashing functions ($k$). The values chosen for these values were:

- Bloom filter bit array size – $2^{12}$ and $2^{16}$
- Bloom filter hashing functions – 4 and 8

These Bloom filter bit arrays size needed to be two to the power of a multiple of four, if the md5sum hash was to be used to conveniently set the index in the Bloom filter, because each hexadecimal in the md5sum hash has a value of $2^4$. The values of $2^{12}$ and $2^{16}$ are a manageable size and with convenient values also chosen for $k$ of 4 and 8, the Bloom filter implementation at 4.4 above had already shown that a low rate of false positives could be achieved, although this might be at a cost in terms of performance.

### 4.7.2 Fragmentation

To manually fragment the search jpeg file within the disk image is not overly difficult using the dd command, but it is tedious and precise. It was decided to create 2 additional images for the 256MB and 2GB images, so that each would be available with an unfragmented file (1 extent) and also with fragmented into 2 extents and into 3 extents.

### 4.8 Conclusion

This chapter has described the implementation of the experiment programs and experiment environment. The experiment programs were written using Python. The design of the local and remote experiment environments was relatively simple. The local PC was already largely configured, and only required some Python configuration, however the machine specification was not very high. The remote virtual PC was successfully set up on request in the Edinburgh Napier University School of Computing cloud, using a system image created in VMWorkstation on the the local PC. This image ran successfully, with only one noticeable but short interruption to system availability of the cloud. The cloud as expected and hoped, proved a good environment for experiment work.
5 Evaluation

5.1 Introduction

This chapter describes the results of the experiments. The results fall into 2 main sections. Firstly, a baseline was established, and cross checked across the local and remote experiment environments. Once this baseline was established, this provided a basis to vary parameters in experiments and evaluate the impact between experiments and against baseline. Multiple iterations of experiments were straightforward to add to the experiment programs, and were not too difficult to analyse from the resulting csv file. As described in the design and evaluation sections, experiment was to demonstrate if a tool could implement a Bloom filter for locating files by sector hashing. Testing would also demonstrate the effect on speed and accuracy of file location of changing Bloom filter parameters and of file fragmentation.

5.2 Baseline Testing

Baseline timings were first established. The aim of this was to identify that timings were consistent in the experiment environment, and what differences would arise from how the experiments were carried out as opposed to varying parameters such as Bloom filter settings and file fragmentation. For instance the difference between the results of experiment on the local PC or the VM.

5.2.1 IDLE (Python Development Environment)

The first baseline experiment was done on the local PC using the 3 experiment programs. Testing was carried out with no other applications open and with no internet connection, in order to minimise any interference from other applications. This experiment was carried out by running programs in the Python Integrated Development Environment (IDLE).

<table>
<thead>
<tr>
<th>Sampling interval (sectors)</th>
<th>Cycles</th>
<th>Database match</th>
<th>Bloom filter match</th>
<th>Bloom filter match c/w checks &amp; measures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>20.50</td>
<td>18.93</td>
<td>23.86</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>19.30</td>
<td>18.96</td>
<td>20.30</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>10.48</td>
<td>10.02</td>
<td>10.78</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>5.38</td>
<td>5.25</td>
<td>5.73</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>5.49</td>
<td>4.87</td>
<td>7.56</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>5.78</td>
<td>4.46</td>
<td>11.17</td>
</tr>
</tbody>
</table>

Local PC. Bloom filter m/k settings $2^{16}$ /4. Image file = 256Mjusb1

Table 6: Baseline experiment (IDLE) 256MB - Database match vs Bloom filter match
These charts are for unweighted averages, that is Fig 22 is the average from the three different sampling intervals and Fig 23 is the average from the three different number of experiment cycles. The sampling interval has a direct inverse relationship to the number of sectors sampled and so it was expected that with sampling interval increasing time would reduce. It was also likely that this relationship would be linear. This inverse linear relationship is apparent in Fig 22. What is also clear that there is little difference in the overall total programme time between the 3 programs, that is the database match, the Bloom filter match, and the Bloom filter match with detailed metrics. However, the database match was consistently slightly slower.

Increasing the number of cycles should give a better guide to the average values, as it will involve more experiments and have a smoothing effect. Increasing the cycle size reduced the average times slightly. This can only be an averaging effect, and the experiments were identical. The reduction was slight, and suggested that for experiment purposes 10 or 20 cycle experiments could give meaningful comparisons. The ratio of maximum time to average time varied from 1.08 to 2.14, while the ration of minimum time to average time varied from 1.02 to 1.18. That maximum times could vary more is not unexpected. There will be a floor to the minimum time, as the program completes. However, there is not such a clear ceiling, and other processes could interrupt and delay the program. So maximum times can have greater variation.

The experiment suggested that neither program type or experiment cycle size would make much difference to total time. Sampling interval would affect time as expected. The same experiments were then tested with the same settings, but this time instead of running the program within IDLE, the program was run from the command prompt. This might lower resource overhead and might give quicker and more consistent times.
### 5.2.2 Python Command line

<table>
<thead>
<tr>
<th>Command line 256MB (all times in s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling interval</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 7: Baseline experiment (command line) 256MB - Database match vs Bloom filter match

Timings were quicker and were more consistent. The ratio of maximum time to average time ranged from 1.02 to 1.59, and the ratio of minimum time to average time ranged from 1 to 1.02. While maximum times again varied more than minimum times, the range of variation for both maximum and minimum times was lower for both. Also, the average time was much more consistent across different cycle size. Directly comparing the ratio of the times from the command line, with the times from IDLE showed that apart from a few maximums, the command line time was always faster.
Command line/ IDLE time ratio 256MB (all times in s)

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>Cycles</th>
<th>Database match</th>
<th>Bloom filter match</th>
<th>Bloom filter match c/w checks &amp; measures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>500</td>
<td>10</td>
<td>0.83</td>
<td>0.91</td>
<td>0.77</td>
</tr>
<tr>
<td>20</td>
<td>0.89</td>
<td>0.94</td>
<td>0.86</td>
<td>0.84</td>
</tr>
<tr>
<td>100</td>
<td>0.92</td>
<td>0.93</td>
<td>0.92</td>
<td>0.89</td>
</tr>
<tr>
<td>1000</td>
<td>0.85</td>
<td>0.88</td>
<td>0.86</td>
<td>0.85</td>
</tr>
<tr>
<td>20</td>
<td>0.85</td>
<td>0.93</td>
<td>0.76</td>
<td>0.84</td>
</tr>
<tr>
<td>100</td>
<td>0.97</td>
<td>0.95</td>
<td>0.98</td>
<td>0.92</td>
</tr>
<tr>
<td>2000</td>
<td>0.88</td>
<td>0.84</td>
<td>1.21</td>
<td>0.81</td>
</tr>
<tr>
<td>20</td>
<td>0.81</td>
<td>0.90</td>
<td>0.67</td>
<td>0.86</td>
</tr>
<tr>
<td>100</td>
<td>0.77</td>
<td>0.98</td>
<td>0.47</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Local PC. Bloom filter m/k settings 2^16/4. Image file = 256Mjetsbf1

Table 8: Baseline experiment (command line) 256MB - IDLE vs command line

As times were quicker, and more consistent using the command line, it was decided to experiment on that basis, and that cycle sizes as low as 10 could be adequate.

5.2.3 Attached USB Flash Drive

For practicality experiment was going to be made against disk images, that is files which were a bit copy or image of a disk would be examined as if they were a disk. This would work, but would not be realistic for any eventual functional tool because in practice it would be more likely to be examining a disk directly. To see how the speed of processing a disk image on file compared with processing a disk directly, an experiment was made comparing accessing a USB memory stick attached to the PC compared with accessing an image of the USB memory stick on file in an image created using dd.

Command line 256MB (all times in s) (20 iterations)

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>DD or USB</th>
<th>Bloom filter match c/w checks &amp; measures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
</tr>
<tr>
<td>500</td>
<td>DD</td>
<td>16.07</td>
</tr>
<tr>
<td></td>
<td>USB</td>
<td>19.05</td>
</tr>
<tr>
<td>1000</td>
<td>DD</td>
<td>7.71</td>
</tr>
<tr>
<td></td>
<td>USB</td>
<td>10.11</td>
</tr>
<tr>
<td>2000</td>
<td>DD</td>
<td>4.28</td>
</tr>
<tr>
<td></td>
<td>USB</td>
<td>4.73</td>
</tr>
</tbody>
</table>

Local PC. Bloom filter m/k settings 16/4. Image file = 256Mjetsbf1

Table 9: Baseline experiment (command line) 256MB - DD disk image vs attached USB device

This showed that with the local PC setup, processing the image on disk was always quicker than processing the actual attached USB memory stick. However, the local PC did use USB1.0, which is now dated and slow. With more hardware options the speed of direct physical access compared with images on disk could be investigated.

5.2.4 Disk Time and Match Time

Timing comparisons had been on total program times to complete. It was decided to investigate the breakdown of program times. It had already been established that the Bloom
filter program with detailed metrics gave similar overall times to the plain Bloom filter program, so it was reasonable to think that the timing routines in the program were not unduly affecting what they measured and gave a true reflection of program performance. The program has 2 main components which are likely to take time. The I/O process or *disk time* where sectors are sampled and read off the disk, and hashes generated for the sector, and the *match time*, where that hash is used to generate Bloom filter indices which are then used to see if that sector matches against the search file IMG_0036.jpg

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>Bloom filter match c/w checks &amp; measures</th>
<th>Disk times</th>
<th>Match times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Min</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>17.5792</td>
<td>17.2202</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>8.7934</td>
<td>8.6020</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>4.3903</td>
<td>4.2863</td>
</tr>
</tbody>
</table>

Local PC. Bloom filter m/k settings 2^16/4. Image file = 256Mjcusb1

Table 10: Baseline experiment (command line) 256MB - Disk times vs Match times

It is very apparent that while both times are inversely proportional to the sampling interval that the disk times are much more significant in overall time than the match time. This would explain why the Bloom filter times overall were only slightly quicker than the database times. The Bloom filter match would be expected to be much quicker than the database match, and while this showed in the overall time, the effect was diluted by the disk times which were common code for both the Bloom filter program and the database program. The disk times were then analysed further, with timings being compared for reading sectors and making the MD5 hash, compared with just reading the sectors. This was a separate counter program. Without the MD5 hash, then no Bloom filter match was possible. The only purpose of this counter program is to differentiate the times taken within disk time for reading sectors and for hashing sectors.

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>Bloom filter match c/w checks &amp; measures</th>
<th>Disk time – DD read &amp; MD5SUM hash</th>
<th>Disk time – DD read only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Min</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>4.0176</td>
<td>3.8379</td>
</tr>
</tbody>
</table>

Local PC. Bloom filter m/k settings 2^16/4. Image file = 256Mjcusb1

Table 11: Baseline experiment (command line) 256MB - dd times and md5sum times

This showed that within disk time, most of the time is taken by the dd command reading the individual sectors, rather than the MD5 hash command. This is all on the local PC using a disk image on file, but in this case then to improve performance a faster disk which might speed up disk read would have more impact than a faster CPU which could speed up MD5 hashing. Although the match time has already been identified as the lesser component of the overall time, it was investigated further, to give a direct comparison between the database and Bloom filter match times, a comparison which had already been inferred from the overall times. In addition, because the programming language being used, Python, is an interpreted language, it was decided to rewrite the code to use Cython, which allows Python code to be compiled.
5.2.5  Python and Cython

<table>
<thead>
<tr>
<th>Command line 256MB (all times in s) (20 iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling interval</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
</tbody>
</table>

Local PC. Bloom filter m/k settings 2^16 /4. Image file = 256Mjcusbf1

Table 12: Baseline experiment (command line) 256MB - Python vs Cython

The inferred huge speed advantage of the Bloom filter over the Database was confirmed. However, the database used, SQLite, might be low performance, and comparison with another database, such as MySQL could be useful. Cython is quicker than Python, but the speed gains are limited. This experiment suggests that in practice, the main constraint on achieving fast times might still be disk performance.

5.2.6  Local PC vs Remote virtual PC

All the baseline experiment so far had been done on the local PC. While this has a low specification it is a relatively controlled environment, and does not share resources with other users. It was expected that performance would be better on the remote VM, and that most actual experiment, especially on the 2GB file, would be done there. To establish baselines, comparable experimentation was then carried out on the local PC and the remote VM.

<table>
<thead>
<tr>
<th>Command line 256MB (all times in s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling interval</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Bloom filter m/k settings 2^16 /4. Image file = 256Mjcusbf1

Table 13: Baseline experiment (command line) 256MB - Local PC vs remote PC
<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>Cycles</th>
<th>Remote VM Bloom filter match c/w checks &amp; measures</th>
<th>Average</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>10</td>
<td>0.36</td>
<td>0.36</td>
<td>0.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.36</td>
<td>0.37</td>
<td>0.32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.35</td>
<td>0.36</td>
<td>0.29</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>0.36</td>
<td>0.36</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.35</td>
<td>0.36</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.36</td>
<td>0.37</td>
<td>0.36</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>0.32</td>
<td>0.36</td>
<td>0.23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.35</td>
<td>0.37</td>
<td>0.21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.36</td>
<td>0.37</td>
<td>0.35</td>
<td></td>
</tr>
</tbody>
</table>

Bloom filter m/k settings 16/4. Image file = 256Mjcusbf1

Table 14: Baseline experiment (command line) 256MB - comparison local PC vs remote PC

As was expected, because of the higher specification, the remote VM provided better performance than the local PC, and this advantage was very consistent in the time tested, with the ratio average times on the remote VM compared to the local PC varying from 0.32 to 0.36, and in 8 of the 9 experiments the ratio varied from 0.35 to 0.36. To confirm this time advantage, and to investigate how consistent remote VM performance was, the experiment program was set to run continually over an extended period using crontab.

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>Cycles</th>
<th>Remote VM Bloom filter match c/w checks &amp; measures</th>
<th>Average</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>100</td>
<td>3.85</td>
<td>3.46</td>
<td>11.39</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>1.95</td>
<td>1.74</td>
<td>7.88</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>100</td>
<td>0.95</td>
<td>0.87</td>
<td>3.07</td>
<td></td>
</tr>
</tbody>
</table>

Bloom filter m/k settings 2^16/4. Image file = 256Mjcusbf1

Table 15: Baseline experiment (command line) 256MB - Remote PC extended crontab execution

The timings running the program via crontab were significantly quicker than when running the program from the command line while logged on remotely. This result was unexpected and unlooked for. The program times were plotted against the 24 hour clock.
Figure 26: Baseline experiment (command line) 256MB - remote PC program times vs time of day

The higher program times correspond to times of the day when a connection was in place from the local PC to the remote VM. This suggests that resources are used for sustaining a link to the remote session, or performance is hindered due to some interaction with the PC making the connection. When disconnected from the remote VM, but still with a session running on the remote VM, then the program times were much better and are consistent. To check these timings and to see if they might be related to local PC activity, experiments were repeated the following Sunday through to Monday. On Sunday around 1000 to 1100 experiment was done using crontab and command line execution while connected from the local PC and with the local PC busy. At around 2000 to 2030 the connection was repeated from the local PC and experiment repeated but with the local PC relatively quiet.
Again the times are over two days so wrap around but it is very clear that times for the programs to complete on the remote VM are higher when the local PC is connected, and are much higher when that local PC is busy. Whether this is a consequence of issues on the local PC, the remote VM, or both would need further investigation, including accessing the remote VM from a different local PC. However, the consistent and good performance achieved on the remote VM using crontab in a running session, but without the local PC connected, suggests the problem is more to do with the local PC even though the effect is to slow down program execution on the remote VM.

The conclusion for the experiment is that consistent, reliable results on the remote VM will only follow from experiments run while the local PC is disconnected. For minimum and averages times, the times were always quicker running the program at the command prompt rather than within IDLE. Minimum and average times were reasonably consistent across cycle range, and 10 cycles seems an adequate number. Maximum program time is the time which varies the most, and this may be due to interruptions during processing, perhaps due to background processes running on the system. The match using database is always slower than the match using Bloom filter. Despite the overhead, the Bloom filter match supplemented by database check and metrics is faster than the Bloom filter match in 6 out of 18 experiments. The Bloom filter only match is usually fastest, although the time differences are not large. When experimenting with the attached USB memory stick, times were slower than when experimenting using a disk image on file. Although an attached USB memory stick is more realistic, the times will be affected by the relatively slow USB1.0 connection in use.

### 5.3 Experiment Results and Analysis

Baseline experiment had established a consistent experiment environment for experimentation – using crontab to run the Bloom filter with metrics program on the remote VM. It had also established actual times. Experimentation would now vary selected parameters (Bloom filter m and k values, and file fragmentations) to see what effect those had. First as a sanity check comparable figures to earlier comparable baseline experiment were obtained. Times are shown, with the ratio compared to the earlier times in brackets. This was to confirm that the
environment for experiment had not changed from when the baseline experiment was carried out earlier.

<table>
<thead>
<tr>
<th>Command line 256MB crontab (all times in s) (Averaged over 4 repeats over 17 hrs. on each of 3 256MB files)(240 experiments in total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling interval</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>Bloom filter m/k settings</td>
</tr>
</tbody>
</table>

Table 17: Baseline experiment (command line) 256MB - second base lining sanity check

The ratios for the averages are very close, but there is a noticeable difference. This comparison is 240 experiments in total, compared with 3300 earlier. This suggests that more experiments might be needed for truer times. However, as a sanity-check times are close enough not to think that experiment environment changed materially from that used to establish the baseline.

### 5.3.1 Bloom Filter Experiments

To study the effect of changing Bloom filter parameters on accuracy and speed, and how false positive rates compare with actual rates.

Variable parameters in experiment.

- Bloom filter bit array size – $2^{12}$ and $2^{16}$
- Bloom filter number of hashing functions – 4 and 8
- Disk image file size – 256MB and 2GB
- Sampling intervals 500, 1000, 2000, 4000

Thirty-two different combinations of parameters were to be tested. Each experiment was to be repeated over 20 cycles. All experiments were to be run on the VM as crontab job.

Over a range of sampling intervals, the effect on overall program time for different values of m (Bloom filter bit array length) and k (the number of Bloom filter hash functions) was tested. The values for m and k were $2^{16}$ and $2^{12}$, and 8 and 4 respectively, giving four m/k combinations of $2^{16}$/8, $2^{16}$/4, $2^{12}$/8 and $2^{12}$/4. Average and minimum times are given, along with standard deviation.
### Effect of varying m and k on Program Times

#### 256MB Program Times

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>m/k 2^16/8</th>
<th></th>
<th></th>
<th>m/k 2^16/4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
</tr>
<tr>
<td>500</td>
<td>8.36</td>
<td>3.42</td>
<td>3.07</td>
<td>3.76</td>
<td>3.23</td>
<td>0.82</td>
</tr>
<tr>
<td>1000</td>
<td>2.56</td>
<td>1.57</td>
<td>1.02</td>
<td>2.06</td>
<td>1.61</td>
<td>0.60</td>
</tr>
<tr>
<td>2000</td>
<td>1.12</td>
<td>0.81</td>
<td>0.62</td>
<td>0.88</td>
<td>0.84</td>
<td>0.13</td>
</tr>
<tr>
<td>4000</td>
<td>0.51</td>
<td>0.43</td>
<td>0.20</td>
<td>0.47</td>
<td>0.44</td>
<td>0.08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>m/k 2^12/8</th>
<th></th>
<th></th>
<th>m/k 2^12/4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
</tr>
<tr>
<td>500</td>
<td>4.31</td>
<td>4.10</td>
<td>0.28</td>
<td>3.91</td>
<td>3.79</td>
<td>0.08</td>
</tr>
<tr>
<td>1000</td>
<td>2.15</td>
<td>2.09</td>
<td>0.16</td>
<td>1.97</td>
<td>1.95</td>
<td>0.02</td>
</tr>
<tr>
<td>2000</td>
<td>1.06</td>
<td>1.01</td>
<td>0.08</td>
<td>0.99</td>
<td>0.98</td>
<td>0.01</td>
</tr>
<tr>
<td>4000</td>
<td>0.52</td>
<td>0.49</td>
<td>0.02</td>
<td>0.50</td>
<td>0.49</td>
<td>0.01</td>
</tr>
</tbody>
</table>

#### 2GB Program Times

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>m/k 2^16/8</th>
<th></th>
<th></th>
<th>m/k 2^16/4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
</tr>
<tr>
<td>500</td>
<td>62.27</td>
<td>25.27</td>
<td>21.20</td>
<td>35.66</td>
<td>24.40</td>
<td>9.48</td>
</tr>
<tr>
<td>1000</td>
<td>24.58</td>
<td>12.17</td>
<td>11.47</td>
<td>16.58</td>
<td>12.19</td>
<td>4.70</td>
</tr>
<tr>
<td>2000</td>
<td>13.90</td>
<td>6.03</td>
<td>8.51</td>
<td>8.70</td>
<td>6.43</td>
<td>3.95</td>
</tr>
<tr>
<td></td>
<td>5.84</td>
<td>3.01</td>
<td>3.37</td>
<td>4.42</td>
<td>3.18</td>
<td>2.22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>m/k 2^12/8</th>
<th></th>
<th></th>
<th>m/k 2^12/4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
<td>Average</td>
<td>Min</td>
<td>σ</td>
</tr>
<tr>
<td>500</td>
<td>37.43</td>
<td>31.08</td>
<td>6.25</td>
<td>32.74</td>
<td>28.62</td>
<td>2.98</td>
</tr>
<tr>
<td>1000</td>
<td>19.45</td>
<td>16.45</td>
<td>4.54</td>
<td>17.52</td>
<td>14.49</td>
<td>3.86</td>
</tr>
<tr>
<td>2000</td>
<td>10.66</td>
<td>7.85</td>
<td>3.39</td>
<td>8.76</td>
<td>7.52</td>
<td>2.47</td>
</tr>
<tr>
<td>4000</td>
<td>4.78</td>
<td>3.76</td>
<td>1.53</td>
<td>4.36</td>
<td>3.73</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 18: 256MB - Comparison of various Bloom filter m and k values – program times

Times reduce with increasing sampling interval, and so does standard deviation as the larger number of samples has a smoothing effect.
Comparing the different values of $m$ and $k$, these have a little clear effect on overall program time. The glaring exception to this small difference is the average times for the 500 sampling interval with $m$ and $k$ settings of $2^{16}$ and 8. These times were markedly higher. This was unexpected and was not immediately explicable. One explanation was that these experiment had been affected by an external factor – such as resource conflicts on the remote VM. This is the simplest explanation and to test for this, the experiments were repeated in part. With $m$ and $k$ settings of $2^{16}$ and 8, and $2^{16}$ and 4, a cycle of 100 experiments was run on Sunday 22/7.
This time, the $2^{16}$ and 8 times were virtually identical to the $2^{16}$ and 4 times. Accordingly, it is reasonable to assume that the original $2^{16}$ and 8 times for the 500 sampling interval were an anomaly due to external factors affecting the average time and not significant. This is also suggested by the original minimum times which were very similar for all values of m and k at all sampling intervals. That was for the overall program times. A similar study was done of the matching times. Disk times do not depend on the Bloom filter so would not be affected by the values of m and k, so any differences in time due to m and k settings will be most apparent in the match times.

**Effect of varying m and k on Match Times**

<table>
<thead>
<tr>
<th>256MB Match Times (in s) for various m/k combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>m/k $2^16/8$</td>
</tr>
<tr>
<td>Sampling interval</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>4000</td>
</tr>
</tbody>
</table>

| m/k $2^12/8$ | m/k $2^12/4$ |
| Sampling interval | Average | Min | σ | Average | Min | σ |
| 500 | 0.69795 | 0.66138 | 0.02615 | **0.41455** | 0.38968 | 0.01335 |
| 1000 | 0.35135 | 0.32975 | 0.01700 | **0.20900** | 0.19480 | 0.00950 |
| 2000 | 0.17523 | 0.16158 | 0.00928 | **0.10468** | 0.09075 | 0.00705 |
| 4000 | 0.08548 | 0.07623 | 0.00573 | **0.05258** | 0.04333 | 0.00505 |

Table 20: 256MB - Comparison of various Bloom filter m and k values – match times

<table>
<thead>
<tr>
<th>2GB Match Times (in s) for various m/k combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>m/k $2^16/8$</td>
</tr>
<tr>
<td>Sampling interval</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>4000</td>
</tr>
</tbody>
</table>

| m/k $2^12/8$ | m/k $2^12/4$ |
| Sampling interval | Average | Min | σ | Average | Min | σ |
| 500 | **6.2105** | 5.6929 | 0.7195 | **3.39600** | 3.16220 | 0.16930 |
| 1000 | **3.1364** | 2.8009 | 0.4432 | **1.74930** | 1.56220 | 0.24747 |
| 2000 | **1.6177** | 1.3846 | 0.2406 | **0.86067** | 0.75323 | 0.12883 |
| 4000 | **0.7674** | 0.6872 | 0.0991 | **0.42527** | 0.38107 | 0.04217 |

Table 21: 2GB - Comparison of various Bloom filter m and k values – match times
While match times remain the lesser component of overall program times, when studying the effect of m and k, the effect is most noticeable by looking at the match times. The match times for m and k values of $2^{12}$ and 8 are highest, followed by times for $2^{16}$ and 4, with times for $2^{16}$ and 8 slightly higher than times for $2^{16}$ and 4. These differences can be explained as illustrated below. The extra time expected to calculate 8 hashing functions compared to 4 was not noticeable. However, with m at $2^{12}$ the number of false-positives was so much greater than with m at 16 that program time was taken in recording and checking all these false-positives, each of which was checked in the database.

Accuracy and Hit Rate

---

Figure 33: 256MB various Bloom filter values – average match time vs sampling interval

Figure 34: 2GB various Bloom filter values – average match time vs sampling interval

Figure 35: 256MB various Bloom filter values – Min. match time vs sampling interval

Figure 36: 2GB various Bloom filter values – Min. match time vs sampling interval
Accuracy is being defined as the proportion of positives which are true positives. So, if the Bloom filter matches are all true positives, then accuracy is 1, the maximum possible. If none of the Bloom filter matches are true positives, that is they are all false positives, then accuracy is 0, the minimum possible. As previously identified in the Bloom filter design, a value of $m$ of $2^{12}$ gives a very high number of false positives. Each of these false positives is tested and identified as such in the program, and the overhead of that check will be why the match times for a value of $m$ of $2^{12}$ are much higher than those for a value of $m$ of $2^{16}$.

Hit rate is defined as the proportion of experiments which detect at least one true positive. So, if every experiment detected at least one true positive then hit rate would be 1, the maximum
possible. That corresponds to every experiment detecting the file being searched for. Conversely a hit rate of 0, the minimum possible, would correspond to no experiment detecting the file being searched for. Increasing time corresponds to increased number of sectors being sampled, so it would be expected that this would see an increase in hit rate. Hit rates are determined by sampling interval, not Bloom filter settings, but what is apparent here from Fig 38 is that with m at $2^{16}$ times are quicker than with m at $2^{12}$, and that also with m at $2^{16}$ then hit rate can be increased with little effect on match time, though because there will be additional disk time there would be a noticeable effect on overall time. To differentiate in more detail any difference in timings for m and k values of $2^{16}$ and 8 and $2^{16}$ and 4, experiments were repeated at 12 noon Friday 20/7, 20 cycles, with sampling intervals of 50 over the range 1700 to 4000. This will give increased granularity of data over the range where hit rate is less than 1.

Figure 39: 256MB various Bloom filter values – Hit rate vs average match time

The times involved are very small, but this confirms that significant hit rates do not require a significant increase in match time. Overall program time can also be mapped to give a measure of the overall impact of increasing the hit rate.

Figure 40: 256MB various Bloom filter values – Hit rate vs average prog. time

As would be expected, with k at 4 rather than 8 times were quicker as fewer hash functions had to be generated, although the effect appears very slight. So, in the experiment, the best times were obtained with m at $2^{16}$ rather than $2^{12}$ and that difference was significant,
and with k at 4 rather than 8, though that difference is slight. The effect of varying m and k on accuracy and hit rate was then analysed further.

### 256MB True Positive and Accuracy

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>m/k $2^{16}$/8</th>
<th>m/k $2^{16}$/4</th>
<th>m/k $2^{12}$/8</th>
<th>m/k $2^{12}$/4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hit Rate</td>
<td>Av. Hits</td>
<td>Hit Rate</td>
<td>Av. Hits</td>
</tr>
<tr>
<td>500</td>
<td>1</td>
<td>3.40</td>
<td>1</td>
<td>3.43</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1.70</td>
<td>1</td>
<td>1.79</td>
</tr>
<tr>
<td>2000</td>
<td>0.85</td>
<td>0.85</td>
<td>1</td>
<td>0.89</td>
</tr>
<tr>
<td>4000</td>
<td>0.48</td>
<td>0.48</td>
<td>1</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Table 22: 256MB various Bloom filter values – True positives and accuracy

**Note:** 0.00 is zero to 2 decimal places, 1 is integer one.

### 2GB True Positive and Accuracy

<table>
<thead>
<tr>
<th>Sampling interval</th>
<th>m/k $2^{16}$/8</th>
<th>m/k $2^{16}$/4</th>
<th>m/k $2^{12}$/8</th>
<th>m/k $2^{12}$/4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hit Rate</td>
<td>Av. Hits</td>
<td>Hit Rate</td>
<td>Av. Hits</td>
</tr>
<tr>
<td>500</td>
<td>1</td>
<td>3.40</td>
<td>1</td>
<td>3.43</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1.67</td>
<td>1</td>
<td>1.65</td>
</tr>
<tr>
<td>2000</td>
<td>0.83</td>
<td>0.83</td>
<td>1</td>
<td>0.85</td>
</tr>
<tr>
<td>4000</td>
<td>0.42</td>
<td>0.42</td>
<td>1</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Table 23: 2GB various Bloom filter values – True positives and accuracy

**Note:** 0.00 is zero to 2 decimal places, 1 is integer one.
The effect of varying \( m \) and \( k \) on the hit rate is very slight, and the hit rate depends far more upon sampling interval, although once the hit rate is one, then increasing the sampling interval has no gain. For accuracy, then accuracy is much higher with \( m = 2^{16} \) than with \( m = 2^{12} \), and this leads to far fewer false positives and a quicker match time. With \( m = 2^{16} \) then increasing \( k \) from 4 to 8 has a significant increase in accuracy of about 10\%. This all accords with the results of the initial evaluation of Bloom filters for accuracy and hit rate, supplemented by the observed beneficial effect on speed of higher accuracy. So, optimising \( m \) and \( k \) for accuracy has the additional benefit of increasing speed. In the experiment it is clear that a Bloom filter bit array length of \( 2^{16} \) gives a much better result in accuracy than a bit array length of \( 2^{12} \). Whether the number of Bloom filter hashing functions is 4 or 8 makes little difference.

### 5.3.2 Fragmentation Experiments

Fragmenting files should have no effect on speed of analysis, but might have an effect on accuracy rate compared with the same sampling rate for unfragmented files. A comparison could also be made between the predicted accuracy rates and the actual rates measured. Rates were predicted on the simple model based on file fragments described in 2.9.2 above, as well as a prediction based on random sectors as described in 2.9 above.
The variable parameters in experiment would be;

- Target file size – 256MB or 2GB
- Fragmentation – 1, 2 or 3 fragments (and corresponding maximum fragment size)
- Bloom filter bit array size – $2^{16}$ with filter hashing functions – 4 or 8
- Sampling intervals 500, 1000, 2000, or 4000 sectors

This gives 32 different combinations of parameters to be tested, and each experiment would be repeated over several cycles to average out fluctuations. All the experiments would be run in the virtual environment as a crontab job. There were no false-positives due to indistinct sectors, all false-positives were Bloom filter false-positives. Due to rounding errors applying the ratio to the actual hit rates below might sometimes give an apparent predicted hit rate of greater than one, but this is a rounding error in the table values.

### Accuracy and Hit Rate

#### 256MB True Positive and Accuracy (80 experiments)

<table>
<thead>
<tr>
<th>m/k</th>
<th>Frags</th>
<th>Interval sector</th>
<th>True Positive Hit Rate</th>
<th>Accuracy</th>
<th>Av. Hits</th>
<th>Ratio Predicted/Actual Hit Rate fragments</th>
<th>random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16/8</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>3.40</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.70</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.85</td>
<td>0.85</td>
<td>1</td>
<td>1.03</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.48</td>
<td>0.48</td>
<td>1</td>
<td>0.93</td>
<td>2.15</td>
</tr>
<tr>
<td>16/8</td>
<td>2</td>
<td>500</td>
<td>1</td>
<td>3.49</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.76</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.75</td>
<td>0.75</td>
<td>1</td>
<td>0.94</td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.38</td>
<td>0.38</td>
<td>1</td>
<td>1.09</td>
<td>2.79</td>
</tr>
<tr>
<td>16/8</td>
<td>3</td>
<td>500</td>
<td>1</td>
<td>3.41</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.78</td>
<td>1</td>
<td>0.92</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.85</td>
<td>0.85</td>
<td>1</td>
<td>0.75</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.39</td>
<td>0.39</td>
<td>1</td>
<td>1.03</td>
<td>2.77</td>
</tr>
<tr>
<td>16/4</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>3.43</td>
<td>0.97</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.79</td>
<td>0.97</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.89</td>
<td>0.89</td>
<td>0.95</td>
<td>1.00</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.39</td>
<td>0.39</td>
<td>1.00</td>
<td>1.15</td>
<td>2.66</td>
</tr>
<tr>
<td>16/4</td>
<td>2</td>
<td>500</td>
<td>1</td>
<td>3.35</td>
<td>0.96</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.66</td>
<td>0.96</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.90</td>
<td>0.90</td>
<td>0.99</td>
<td>0.98</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.46</td>
<td>0.46</td>
<td>0.98</td>
<td>0.89</td>
<td>2.28</td>
</tr>
<tr>
<td>16/4</td>
<td>3</td>
<td>500</td>
<td>1</td>
<td>3.51</td>
<td>0.98</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.71</td>
<td>0.96</td>
<td>0.92</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.85</td>
<td>0.85</td>
<td>0.99</td>
<td>0.75</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.45</td>
<td>0.45</td>
<td>1</td>
<td>0.87</td>
<td>2.34</td>
</tr>
</tbody>
</table>

Table 24: 256MB - Comparison of various fragmentation – True positives and accuracy
### 2GB True Positive and Accuracy (60 experiments)

Comparing the results is easier when they are charted.

<table>
<thead>
<tr>
<th>m/k</th>
<th>Frags.</th>
<th>Interval sector</th>
<th>True Positive</th>
<th>Accuracy</th>
<th>Ratio Predicted/Actual Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hit Rate</td>
<td>Av. Hits</td>
<td>fragments</td>
</tr>
<tr>
<td>16/8</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>3.40</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.67</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.83</td>
<td>0.83</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.42</td>
<td>0.42</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>1</td>
<td>1</td>
<td>3.32</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.80</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.78</td>
<td>0.78</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.40</td>
<td>0.40</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>500</td>
<td>1</td>
<td>1</td>
<td>3.42</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.70</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.90</td>
<td>0.90</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.53</td>
<td>0.53</td>
<td>1</td>
</tr>
<tr>
<td>16/4</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>3.43</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>1</td>
<td>1.65</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>0.85</td>
<td>0.85</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>0.48</td>
<td>0.48</td>
<td>0.92</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3.40</td>
<td>0.93</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1.68</td>
<td>0.90</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.88</td>
<td>0.88</td>
<td>0.91</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.43</td>
<td>0.43</td>
<td>0.94</td>
<td>0.93</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.42</td>
<td>0.90</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1.68</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.90</td>
<td>0.90</td>
<td>0.92</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.45</td>
<td>0.45</td>
<td>0.90</td>
<td>0.83</td>
</tr>
</tbody>
</table>

*Table 25: 2GB - Comparison of various fragmentation – True positives and accuracy*
The hit rate is similar for the 256MB and 2GB files. The difference depending on fragmentation is very small and insignificant. The factor making a difference is clearly the sampling interval, and as that relates directly and inversely to the number of sectors sampled that is not a surprise.
Figure 48: 2GB - Comparison of various fragmentation – Accuracy vs sampling interval

For Bloom filter bit array length $m$ of $2^{16}$ and Bloom filter number of hashing functions $k$ of 8 then the results for accuracy are congruent with an accuracy of 1, and unaffected by fragmentation. With those respective Bloom filters settings as $2^{16}$ and 4 accuracy is less than 1, but there is no apparent relationship to fragmentation. Accuracy rates for Bloom filter setting $s$ of $2^{16}$ and 4 are higher with the 256MB file than the 2GB file.

Figure 49: 256MB - Comparison of various fragmentation – True positive hit rate vs. accuracy
Plotting the hit rate against accuracy shows that with Bloom filter settings of $2^{16}$ and 8, accuracy is always 1, while the hit rate varies. With the Bloom filter settings of $2^{16}$ and 4, there is no clear relationship between hit rate and accuracy.

**Predicted against Actual Hit Rate**

*Figure 50: 2GB - Comparison of various fragmentation – True positive hit rate vs. accuracy*

'Fragment based predict' means that the hit rate was predicted using the formula in '2.9.2 File Fragmentation'.
'Random based predict' means that the hit rate was predicted using the formula in '2.9.1 Sampling Files on Disk'.

Figure 52: Figure 39: 256MB - Comparison of various fragmentation – Ratio predicted/actual hit rate vs. sampling interval

Figure 53: 2GB - Comparison of various fragmentation – Ratio predicted/actual hit rate (based on fragment size) vs. sampling interval
Comparing the predicted hit rate with the actual hit rate recorded in the experiments, then two results are clear. Most clear is that fragmentation makes no significant difference to hit rate. It is clear from the figures above that the hit rate varies according to sampling interval but is not affected by the level of fragmentation.

The predicted hit rates based on file fragments are close to the actual hit rates measured. The predicted hit rates based on random individual sectors are accurate at the lower sampling intervals, but are inaccurate at the higher sampling interval of 4000 sectors. The explanation for this is not known.

5.4 Conclusion

This chapter has described and analysed the results of testing the hypotheses from the aims and objectives which was identified in the introduction to the design chapter. The experiment procedure went relatively smoothly. There was some iteration of program design, as metrics to record and how best to record them were identified. Summarizing the results to a text file summary gave a useful, readable overview of each test, while the csv file allowed for analysis and charting using a spreadsheet. The experiments were successful and ran many times and logged data. It was clearly demonstrated that a tool to do sector hash analysis using a Bloom filter can be written in Python.

When using a Bloom filter for comparisons the setting of m and k can have a significant, and predictable effect on accuracy. The choice of values for experiment was to some extent initially arbitrary but the comparison of values of m of $2^{16}$ and $2^{12}$, and for k of 8 and 4 showed a very wide range of accuracy especially between the m values for m of $2^{16}$ and $2^{12}$. The results show that in this experiment environment a Bloom filter gives a faster match than a database lookup. This can be achieved with a low, and predicted low, false positive rate by selecting appropriate parameters for the Bloom filter. In this case the key to this was in using the Bloom filter bit length of $2^{16}$ rather than $2^{12}$. Whether the number of Bloom filter hashing functions was 4 or 8 made relatively little difference. In this experiment environment, disk read and hashing is much more significant than database or Bloom filter match time. The time taken for the Bloom filter match was negligible compared to the time taken for the disk read and md5sum hashing.

With fragmented files, as expected, a sampling interval less than the file size, or less than the largest fragment size, guarantees a hit. Any other result would have been a surprise. While the formula for predicting the probability of a hit based on fragment size was validated, this has little practical application because in practice fragmentation is not usually known or predictable. In a real world example file fragmentation would be unknown and unknowable. Fragmentation has no significant measured effect on hit rate or accuracy.
6 Conclusion

6.1 Overall Conclusion

The thesis showed by research of the literature that there is a requirement for improved techniques to help support the faster analysis of data in digital forensic investigations. The example of looking whether a particular file is present on media can arise in several situations. It may be that a device has been seized in an investigation, it may be that there are several suspect devices. It may be that only a short period of time is available for inspection, such as at a border checkpoint. The increase in media storage capacity, and the increase in the use of computers and other electronic devices is all increasing the scale of the task faced in such a situation as looking for a particular file. This creates a backlog of data which has not yet been analysed.

The research identified the use of hashes as a reliable means of identifying file matches, even where a file is renamed. Hashing can be applied at a sector level on disk as well as at a file level. This has several advantages. It supports the identification of files which are incomplete and may have been deleted and had several sectors overwritten. It also supports the technique of sampling sectors from disk, rather than reading the full disk. This has the advantage of accelerating analysis as less data needs to be read, but for the same reason the likelihood of locating a sector from a particular file becomes probability based instead of certainty based as with full disk analysis. This probability can be predicted, and a high level of probability can be achieved using sampling.

The matching of hashes can be achieved using a database and looking up values. An alternative to using a database for matching is to use a Bloom filter. The Bloom filter can offer a significant speed advantage. There is a probability of false positives, but not of false negatives. Similar to the probability of finding a particular file by random sampling of sectors on disk, the probability of Bloom filter false positives can be calculated accurately.

The technique will be as effective against ordinary deleted files because it looks at sectors on the disk, it does not use the file directory information. No technique can work in recovering properly sanitized deleted files, but in practice many tools offering that capability are imperfect and 'deleted' files remain vulnerable to discovery.

6.2 Appraisal of Achievements

The aim of this thesis is to investigate digital forensics investigation triage using the theory behind sampling small block file for hash analysis, and to experimentally test that theory, and investigate the effects of changing the Bloom filter settings and the effect of file fragmentation. Objectives to support this aim are as follows;

1. Investigate the issues in digital forensics investigation triage relating to data collection and examination, in particular around locating files by using sector sampling, hashes, and Bloom filters.

2. Design digital forensics investigation triage tools and design experiments to investigate the effect of varying Bloom filter settings and of file fragmentation on the performance of the tools.

3. Implement prototype tools and an experiment environment. These tools are to include using database and Bloom filter matching.

4. Evaluate and assess the results of the experiment on the tools to assess the effect of varying Bloom filters and of file fragmentation, and identify possible future work.

In support of these objectives the following hypotheses was also specified for testing by experiment.
1. A prototype digital forensics investigation tool can be created using, sampling, sector hash analysis and a Bloom filter.

2. Using a Bloom filter will give a faster match than a database lookup and can have an acceptable false positive rate.

3. The constraint on speed of finding a file on the target is the speed of matching against the Bloom filter, rather than the speed of reading the sectors and hashing them.

4. For contiguous or fragmented files a regularly spaced sampling interval less than the file size or largest fragment size guarantees the file will be located. This can be tested and confirmed.

6.2.1 Objective 1

Investigators digital forensics investigation triage relating to data collection and examination, in particular around locating files by using sector sampling, hashes, and Bloom filters.

The literature review established that there is a real requirement for a digital forensics investigation triage tool to use for data collection and examination. The increasing in data storage capacities and the use of computer and other electronic devices which store data is leading to a backlog of data for analysis, and impeding investigations and checks. A triage tool could implement a combination of techniques, namely sampling sectors from a disk, hashing these sectors and using a database or Bloom filter to match hashes against hash values for a known file. This approach is probability based but the probabilities for selecting a sector with sampling, and for false-positive Bloom filter matches can be predicted. The techniques are not novel but there is an apparent lack of a widely used triage tool to implement them.

6.2.2 Objective 2

Design digital forensics investigation triage tools and design experiments to investigate the effect of varying Bloom filter settings and of file fragmentation on the performance of the tools.

Tool designs and experiment designs to test these hypotheses were outlined. This included a program to populate a database and a Bloom filter using the sector hashes from a file. Tools to use these to check for matches against sectors sampled from a disk were also outlined. A series of experiments were designed to test for the effect of varying Bloom filter parameters, and of file fragmentation. The experiments were to be repeated to give more reliable results than would be achieved from a single, or limited number of, experiments. This iteration was achieved using looping within programs and by running programs regularly using crontab.

The original design worked well in practice. The experiment environments of the local home PC and remote virtual PC were compatible, and programs and data ran equally and consistently on both, with the proviso as identified that performance was better on the remote virtual PC, and that this performance advantage was significantly greater when there was no connection from the local home PC. Disk image files containing a fragmented file were manually created.

6.2.3 Objective 3

Implement prototype tools and an experiment environment. These tools are to include using database and Bloom filter matching.

In the implementation chapter of this thesis the detail program design and experiment environment as implemented were outlined. The programs were written in Python. There
were two experiment environments, a local home PC and a remote virtual PC. Both these environments were functionally alike, with the same OS and programs, but the remote virtual PC offered better performance. A disk image available as a corpora on the internet was downloaded and used to populate a 256MB and 2GB memory stick with data to be used in the experiment. These memory sticks were used to provide disk images on file, which were manipulated to provide disk images with fragmented, as well as unfragmented files. The experiment environments functioned well, and the remote virtual PC proved a convenient experiment environment which could be accessed from different locations, and could be left running the experiment programs unattended. The programs written in Python did, as would be expected, need debugging and refining, but this was all achieved and the programs ran mostly trouble free. The writing to csv file for analysis and charting proved effective, as did the repetition of experiment by using crontab on the remote virtual PC to execute series of programs. With an image still present in the cloud environment, that image could easily be cloned in vmware including the source data and programs, which would allow anyone else to repeat the experiments or carry out variations or new experiments.

**6.2.4 Objective 4**

*Evaluate and assess the results of the experiment on the tools to assess the effect of varying Bloom filters and of file fragmentation, and identify possible future work*

Initial base lining established the effect of experiment environment factors such as whether the experiment programs were runs in the IDLE development environment or at the command line. As well as identifying which factors made a difference it also identified which would make for faster experiment. Repeated baseline experiment also looked for any variations or variability arising from the time of experiment. This was especially important for the remote virtual PC which was in a hosted environment shared by others and it was important to be assured that there were no big changes in performance over time, such as might arise from resource contention. The experiment process ran efficiently, including multiple cycles of experiment run by looping in the programs, and running the programs regularly using crontab. The results of experiment were written to a csv file which allowed analysis later of the results.

The original choice of Bloom filter parameters, with a bit length of $2^{12}$ or $2^{16}$ and number of hashing functions 4 or 8 seems to have been fortunate. While bit length $2^{12}$ gave poor results with a very high number of false positives, the bit length of $2^{16}$ gave a low rate of false positives. The predicted Bloom filter false-positive rates were achieved in practice. Varying the Bloom filter parameters for the bit length of the Bloom filter bit array, and for the number of hashing functions used on setting the Bloom filter did have an effect. This effect was most noticeable in the change between a Bloom filter bit length of $2^{12}$ and $2^{16}$. The Bloom filter false positive rates achieved in experiment matched those predicted in the research, and with the $2^{16}$ bit length the false positive rates were acceptable. The Bloom filter match times were significantly quicker than the database match times, although because the times for reading sectors off disk and hashing the sectors was always much greater than the match times for either the database or Bloom filter, the overall program times were little improved.

With fragmented files, there was has no significant measured effect in the experiments on hit rate or accuracy. The probability of locating a fragmented file by sector sampling could be accurately predicted based on the size of fragments, however this has little practical application because in practice file fragmentation is usually unknown and unknowable.

When reading sectors from a disk, such as with sampling, entries in the file system directory entry are irrelevant. If the data in the sector matches it will be a match. If the file is fragmented then recovery may be difficult, but the definition of a distinct sector should mean that it's presence is proof that the file it contains was present on the disk.
6.2.5   **Hypothesis 1**

A prototype digital forensics investigation tool can be created using, sampling, sector hash analysis and a Bloom filter.

**Proved.** A prototype digital forensics investigation tool which used sector sampling, sector hash analysis and a Bloom filter for matching hashes against hashes for a known file was successfully created. The tool was able to locate the known file on the source disk image file with a level of probability. The probability of locating the file, the hit rate, depends on the number of samples and can be predicted. The accuracy depends on the number of Bloom filter false-positives which can also be predicted. A tool was also created which used a database for matching hashes, which was slower than the Bloom filter match, however the database can be used alongside the Bloom filter to confirm true-positives. The prototype tool is written in Python, but the algorithms could be written in other languages.

6.2.6   **Hypothesis 2**

Using a Bloom filter will give a faster match than a database lookup and can have an acceptable false positive rate.

**Proved.** The evaluation confirmed that the Bloom filter gave a faster match than the database lookup. That is based on this experiment environment, including a SQLite database which may not be the fastest database, however the result validates the research reports of the benefits of Bloom filters. The Bloom filter false positive rate, measured in the experiments as accuracy was as predicted and was acceptable when using a Bloom filter bit array of length $2^{16}$.

6.2.7   **Hypothesis 3**

The constraint on speed of finding a file on the target is the speed of matching against the Bloom filter, rather than the speed of reading the sectors and hashing them.

**Disproved.** In the experiments the disk time to read sectors from disk and hash them was much greater than the match time for either database or Bloom filter. This has implications for developing a practical tool as it implies that the constraint on performance may be disk I/O.

6.2.8   **Hypothesis 4**

For contiguous or fragmented files a regularly spaced sampling interval less than the file size or largest fragment size guarantees the file will be located. This can be tested and confirmed.

**Proved.** Based on sampling interval, for a known file or fragment size then the probability of a hit could be accurately predicted and where the sampling interval is a regular size less than the file or fragment size then a hit is guaranteed and the number of hits can be accurately predicted.

6.3   **Future Work**

6.3.1   **Alternative to Python**

Python proved a convenient programming language and gave results consistent with predictions for Bloom filter settings. Cython gave slightly faster results. Other options could be tried to speed up Python, such as the Psycho just-in-time compiler or the Shedskin C++ translator (Lutz, 2009).

It would be possible to implement the program in another language, such as Java or C#. Speed benefits might be slight, because the main constraint has already been identified as disk
read and hashing, which depends on system performance. However there might be some speed benefits and deployment of a functional tool for common operating systems might be improved, and a program in another language might be a more practical choice on which to base a functional tool for distribution.

### 6.3.2 Alternative Hardware

As disk read and hashing performance had such an impact on performance it would be interesting to do comparisons to see the effect of improving CPU, memory, and disk speed on performance. This could involve using a range of more powerful hardware than was available for this thesis. The experiments in the thesis mostly used disk image files for analysis. It would be more realistic to use attached USB flash drives, and include SD cards. With better testing hardware and faster USB ports it would be interesting to see what the result might be on program times. This would be necessary work for designing any practical tool. A likely scenario in practice would be examining either a computer hard disk or USB flash drive by attaching it to another computer. Another likely scenario would be to examine a computer by attaching a USB flash drive loaded with the SHAFT tool and run the tool from the USB flash drive.

### 6.3.3 A Practical Tool

The prototype tools written for experiment principally were designed to test the sampling algorithm and allow parameters to be varied. The requirements for a functional tool would require more automation and a user interface. For instance, it would be possible to have a tool which could be given a file or folder and automatically create a database and Bloom filter from the disk sector hashes for that file, eliminating likely indistinct sectors. The Bloom filter could be used alone for matching, but the database of file names against hashes could be useful for confirming hits as true positives and identifying the file. Another tweak would be that any tool searching for files could stop as soon as it has found a match, rather than continue to the end of the disk. For disks containing the search file, this would be expected on average to reduce search times by half.

As well as searching for data files, the tool could search for application files, such as encryption software, or steganography software, which might be of interest to a forensics investigator. This could also be used to extend the work already done on identifying the mix of certain files types, such as audio, on a disk (S. Garfinkel et al., 2010).

It is difficult to draw conclusions about the relevance of these timings to real world scenarios because experiment was carried out using a fairly simple example of searching for a single jpeg file, and the hardware used for experiment was relatively low powered. Given that there have been several research implementations of this technique it does seem surprising that it is not routinely used, especially as it addresses a real problem. It may be there are issues, such as performance, or that it is because it uses sampling rather than full disk analysis and so its results are probability based rather than certainty based, but this is speculation. The fact remains that a useful method to help with a real problem does not seem to be routinely used.

### 6.3.4 Network Analysis

Another application which could be investigated would be to apply similar sampling technique to network packet payloads for content. One study concluded that in-line logging for malicious traffic should be restricted because of the impact on the network devices (Graves, Buchanan, et al, 2006), but a sampling based approach looking for content should be practicable as it would reduce the traffic to be sampled.
6.3.5 Encryption

There is a growing problem with full disk encryption with digital information being made inaccessible to the investigator (Casey, Fellows, et al, 2011). Sector sampling in such cases might confirm full disk encryption if discriminators for encryption could be identified. In addition, on an unencrypted disk, sector sampling could be used to identify the presence of applications of interest, which might have an anti-forensics application such as steganography. The presence of such tools in themselves might be suspicious in certain circumstances and might also guide an initial investigation. For example, the presence of steganographic software might lead to checking graphic files more closely.

6.4 Personal Reflection

Computer Science is one of the more recent sciences, certainly very recent compared to the classic science of physics, chemistry and biology. However, it is still a surprise to read references as recent as 2005 (Denning, 2005) which seem to suggest that the status of computer science as a proper science is not assumed.

Although a lot of work has been done by many in establishing and providing standard digital corpora for digital forensic research there is a lack of their wide adoption. This means that comparing the results of one researchers work with another, or the performance of one tool with another can be complicated. As well as digital corpora for media, with the widespread adoption of virtual computing the concept of digital corpora could be extended to included a corpora of virtual machines, for example as vmware images. Although the performance of a virtual machine is partly dependant upon its host, and the available hardware, there is no reason why a range of standard virtual machines, with or without sample data, could not be provided and used for experiment. The availability and sharing of common configurations could improve comparability of different research. These common configurations could support standard models of scenarios, such as find deleted file, find file hidden by encryption and so on.

The concept behind the Bloom filter is inspired and ingenious and deserves to be widely known. The basic concept is not too difficult. The mathematics behind the probability predictions are difficult to absorb, but are easy to apply. It was striking and reassuring how excellent was the match between the predicted accuracy rates with the actual measured rates for various combinations of Bloom filter bit length and number of hashing functions. The proved that the mathematical model matched the implementation in the tool. The chance of that match being coincidental seems impossible, and the only conclusion is that both are working as predicted. Matching with a Bloom filter allows a very rapid lookup against a large collection of reference objects, and is a very powerful tool. The downside of possible high false positive rates can be managed effectively because it can be predicted.

Sector hash matching works. Interestingly the corollary of distinct or unique sectors that is indistinct or common sectors can also be exploited, for example in a picture matching technique which seeks to identify similar images which contain parts in common by matching parts of the image such as a company logo (Lee, Ke, et al, 2010) and in matching similar messages (Pi, Fu, et al, 2009).

The convenience of using the Edinburgh Napier University School of Computing cloud for experiment was significant. From home or university, it was possible to access the same environment, and experiments could be left to run for extended periods unattended. It offered the additional benefit of higher performance than was available on the local PC, while running the same operating system and application.

Two very apparent limitations of the experiment environment reduce the conclusion which can be drawn from experiment as regards how a practical tool would perform in real world situations. First, because disk read and hashing is such a significant constraint it really is necessary to do experiment with varying hardware specifications to see what performance can
be achieved with better but still affordable hardware. Secondly, real world applications would be reading media directly, not from an image file which had been created. So, it would be good to do experiment with reading attached hard drives and memory sticks and see how performance is affected.

With the availability of diverse hardware, the same experiment methods as used in this thesis could be applied. Automated repeated experiment with varying parameters and logging results to csv file for analysis worked well. In this case the metrics in the program did not have a significant impact in slowing program time, but care would be taken to confirm this when experimenting on faster hardware, in case the measuring itself distorted the measurements. This is easy to check for by parallel experiment of programs with and without detailed metrics and comparing overall program times.

From this, and other studies, including at Edinburgh Napier University, it is clear that the theory behind small block file analysis is sound and delivers tangible benefits, and that Bloom filters are excellent for look ups against reference sets. What seems to be missing is a simple, easy to use tool, especially which can be configured by the user to look either for files they have copies of, or use databases distributed by others. It is not clear how effective current tools are, including those which might seem to use this approach already. The convenience of a portable tool which can run off a memory stick is clear. None of the ideas used here are new or novel, what seems to be missing is a simple application. This might suit an open source application and it would be good to develop a tool in collaboration with someone who has a real world use for it, and can assess its value in triaging.
7 References


---

87


Walls, R. J., Learned-Miller, E., & Levine, B. N. (n.d.). Forensic Triage for Mobile Phones with DEC0DE.


Figure 55: Project plan Gantt chart
Student: John Clayton – 95030123
Date: 11 Jul 2012

Objective:
Set up shared dropbox. Add flow diagram.

Progress:
More references added to report but less progress on report drafting than hoped because problems encountered in uploading data to VM for testing. That completed now. Access to VM seems to be fairly reliable.

Supervisor’s Comments:
Organization of project plan is looking very good. Report started and some structure added. Great start. Add to report, fine up structure, start detailing or experts based on lit review so far.
Figure 57: Project diary - week 9

EDINBURGH NAPIER UNIVERSITY
SCHOOL OF COMPUTING
PROJECT DIARY

Student: John Clayton – 95030123
Date: 17 Jul 2012

Supervisor:
Last diary date: 11 July 2012

Objectives:
Share documents in pdf format in Dropbox. Number section headings in report. Rearrange literature review sections. Add more references – aim towards around 80 papers. Each chapter should have an introduction and a conclusion. Add sections covering the practicalities of data acquisition. In literature review give reasons for section and application, explain why its relevant.

Progress:
Arranged a meeting with Mike Dickson. Added references to report and rearranged. Can still add more references, especially from citations in existing references. Nearly completed baseline experiments ready to do main testing in VM. Speed of programs is significantly quicker in the VM environment. Need to plan and design experiments and start evaluation.

Supervisor’s Comments:
Good improvement to the lit review section, with new refs. Great start to evaluation, with presentation of experiment results. Add more background and how to lit reviews, and source experiments.
EDINBURGH NAPIER UNIVERSITY
SCHOOL OF COMPUTING
PROJECT DIARY

Student: John Clayton – 95030123
Date: 24 Jul 2012

Objectives:

- In Literature review group section headings into themes. Consider research methods such as T-tests, and references relevant to that, and to issues discovered with VM performance. Copy report for to Bill Buchanan for sight of experiment design. Proceed to carry out experiments.
- Meet with Mike Dickson.

Progress:

- Met with Mike Dickson for good discussion. Possible addition of Forensic Taxonomy identified by Rich Stacthane. Coding amended to include standard deviation for timings. Also designed to cycle through permutations of parameters for testing. Remote VM access tested from higher spec PC. VM response seems slower during week, but program run times can be compared now with baseline figures for variation. Dissertation end date has now been confirmed as 20th August, not 13th August as originally thought.

Supervisor’s Comments:

- Great progress with Evaluation Section, and analysis to be added discussed well in meeting. Add Analysis to evaluation, and start adding to doing and implementation sections. Add more ad obj’s, instructions to sections; start taxonomy of BIs and function requirements of image for BIs. Add some diagrams to design.
## Appendices

### 9.1 Corpora

<table>
<thead>
<tr>
<th>File name</th>
<th># Sectors</th>
<th># Files</th>
<th>File system [offset] [inode]</th>
<th>Extent1 [# Sectors]</th>
<th>Extent1 [# Sectors]</th>
<th>Extent1 [# Sectors]</th>
<th>Extent1 [# Sectors]</th>
</tr>
</thead>
<tbody>
<tr>
<td>30Mnpsf1.raw</td>
<td>60 799</td>
<td>36</td>
<td>FAT16 [51] [1064]</td>
<td>57312-59035 [1724]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30Mnpsf2.raw</td>
<td></td>
<td></td>
<td>57312-58460 [1149]</td>
<td>48461-49035 [575]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30Mnpsf3.raw</td>
<td></td>
<td></td>
<td>57312-57885 [574]</td>
<td>48461-49035 [575]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256Mjcusbf1.dd</td>
<td>489471</td>
<td>353</td>
<td>FAT16 [257] [40]</td>
<td>57464-59187 [1724]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>57464-58612 [1149]</td>
<td>158613-159187 [575]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>57464-58037 [574]</td>
<td>158613-159187 [575]</td>
<td></td>
<td>258038-258612 [575]</td>
<td></td>
</tr>
<tr>
<td>2Gjcusbf1.dd</td>
<td>3940478</td>
<td>2297</td>
<td>NTFS [63] [99-128-2]</td>
<td>1672639-1674362 [1724]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1672639-1673787 [1149]</td>
<td>2673788-2674362 [575]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All sector numbers are offset relative to the start of the disk image, not the file system. On the NTFS istat reports in blocks of 8 sectors so convert to sector offset by +63 + (8 x blocks), and +7 for last block.
9.2 Code Listings

9.2.1 Makebf.py

```python
# creates Bloom filter and sqlite database of hashes for a reference file
#module: makebf.py
#description: creates Bloom filter and sqlite database of hashes for a reference file
#author: John Clayton, MSc dissertation
#last amended: 15 August 2012
import os, time, sys
from bitarray import bitarray
import sqlite3 as lite

# Bf m is size in bits (must be multiple of 4 , b is conversion to bytes, k is number
# of hash functions
m = 16
b = int(m/float(4))
k = 4

bloomfile = "bloom" + str(m) + "+" + str(k) + ".bin"

# image file, sstart is start sector, sstop is stop sector (+1 as it will define a
# range), sint is interval
ifile = "/home/john/Documents/MScDissertation/images/makebf36.raw"
sstart = 57312
sstop = 59036
distinct = 0

distinct = 0
filename = "IMG_0036.JPG"

# md5sum for indistinct sectors obtained by experiment on raw image
dup = ['bf619eaac0cd3f68d496ea9344137e9b', '890c1babf1a3033bf884d956f24961f',
'0329ca8d3f54d4eb489eda89f9ea56c', '902600c5d0b1268b8ad39328705be438',
'f16bb3da25e4f44b7f8205e724cbff', 'f030c2bfc28d1979156d9a24094936',
'2de018f16a648d7985d21ac2715c9fa', '897b52a357c01337347479a304407388',
'3858e53bf414874716f92f210c90f', '19f8b4bc950a56627b60d31717d3f7e8',
'a4ecdb7e92f5009d5d4543eccad8c390']

start = time.time()
con = lite.connect('hashes.db')
with con:
con.execute('Delete from Hashes')
con.commit()

bf = bitarray(2**m)
for i in range(2**m):
bf[i] = 0
print 'bf initialised to ' , bf

for s in range(sstart,sstop):
c = "dd if=" + ifile + " skip=" + str(s) + " count=1 | md5sum"
result = os.popen(c).read()
if result[:32] not in dup:
distinct = distinct + 1
result = os.popen(c).read()
d = "INSERT INTO Hashes VALUES('%s','%s','%s')" % (s, result[:32], filename)
con.execute(d)
con.commit()

for i in range(k):
result = open(bloomfile, 'wb').read()

print "bf now is " , bf
con.close()

stop = time.time()
print "Bloom filter and Database created ", (stop-start), "s"
print "Distinct = ", distinct
print "Indistinct = ", 1725 - distinct
```

9.2.2 Usedb.py

#module: usedb.py
#description: read sectors and match hashes using reference database
#author: John Clayton, MSc dissertation
#last amended: 15 August 2012
import os, time, datetime, random
import sqlite3 as lite
con = lite.connect('hashes.db')
comment = "usedb"
dbase = "hashes.db"
ofile = "results.txt"
# t is number of test iterations
# image file, sstart is start sector, sstop is stop sector (+1 as it will define a range), sint is interval
# ifile image file location
# 30Mpsf1.raw - f3.raw sstop=60799
# 256Mjcusbf1.dd - f3.dd sstop=489471
# 2Gjcusbf1.dd - f3.dd sstop=3940478
ifile = "/home/john/Documents/MScDissertation/images/256Mjcusbf1.dd"
sint = 2000
sstop = 489471
for j in range(t):
    sstart = random.randint(0,sint-1)
    progstart = time.time()
    matchtime = 0
    with con:
        for s in range(sstart,sstop,sint):
            match = 0
            c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
            os.system(c)
            result=os.popen(c).read()
            matchtime = matchtime - time.time()
            d="Select * FROM Hashes WHERE Hash='%s'" % (result[:32])
            cur = con.cursor()
            cur.execute(d)
            data=cur.fetchone()
            if data:
                print str(j), "db Sector ", s , " hash ", result[:32], " is a match"
                matchtime = matchtime + time.time()
                matchtimelist.append(matchtime)
                progtimelist.append(time.time() - progstart)
                print comment,": ", float(sum(progtimelist))/len(progtimelist) ,min(progtimelist) , max(progtimelist)
            con.close()
    f = open( ofile, 'a', 0)
sampled = int((sstop-sstart+1)/sint)
sample = range(sstart,sstop)
sample = sample[sample.index(random.randint(0,sample))]
progstart = time.time()
with con:
    for s in range(sstart,sstop,sint):
        match = 0
        c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
        os.system(c)
        result=os.popen(c).read()
        matchtime = matchtime - time.time()
        d="Select * FROM Hashes WHERE Hash='%s'" % (result[:32])
        cur = con.cursor()
        cur.execute(d)
        data=cur.fetchone()
        if data:
            sample = sample[sample.index(random.randint(0,sample))]
            matchtime = matchtime + time.time()
            matchtimelist.append(matchtime)
            progtimelist.append(time.time() - progstart)
            print comment,": ", float(sum(progtimelist))/len(progtimelist) ,min(progtimelist) , max(progtimelist)
            con.close()
    f = open( ofile, 'a', 0)
sampled = int((sstop-sstart+1)/sint)
sample = range(sstart,sstop)
sample = sample[sample.index(random.randint(0,sample))]
progstart = time.time()
with con:
    for s in range(sstart,sstop,sint):
        match = 0
        c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
        os.system(c)
        result=os.popen(c).read()
        matchtime = matchtime - time.time()
        d="Select * FROM Hashes WHERE Hash='%s'" % (result[:32])
        cur = con.cursor()
        cur.execute(d)
        data=cur.fetchone()
        if data:
            sample = sample[sample.index(random.randint(0,sample))]
            matchtime = matchtime + time.time()
            matchtimelist.append(matchtime)
            progtimelist.append(time.time() - progstart)
            print comment,": ", float(sum(progtimelist))/len(progtimelist) ,min(progtimelist) , max(progtimelist)
            con.close()
#module: usebf.py
#description: read sectors and match hashes using Bloom filter
#author: John Clayton, MSc dissertation
#last amended: 15 August 2012
import os, time, datetime, random
from bitarray import bitarray
comment = "usebf"
ofile = "results.txt"
# t is number of test iterations
t = 2
progtimelist = []
matchtimelist = []
# BF m is size in bits (must be multiple of 4, b is conversion to bytes, k is number of hash functions
m = 16
b = int(m/float(4))
k = 4
# use one of available Bloom filters
bloomfile = "bloom" + str(m) + "-" + str(k) + ".bin"
# image file, sstart is start sector, sstop is stop sector (+1 as it will define a range), sint is interval
ifile = "/home/john/Documents/MScDissertation/images/2Gjcusbf3.dd"
sint = 512
sstop = 3940478
for j in range(t):
sstart = random.randint(0,sint-1)
progstart = time.time()
matchtime = 0
bf = bitarray()
with open(bloomfile,'rb') as fh:
    bf.fromfile(fh)
for s in range(sstart,sstop,sint):
    match = 0
c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
    os.system(c)
    result=os.popen(c).read()
    matchtime = matchtime - time.time()
    for i in range(k):
        if bf[int(result[i*b:(i*b)+b], 16)] == 0:
            match = 0
            break
        else:
            match = 1
    if match == 1:
        print str(j) , 'bf Sector ' , s , ' hash ', result[:32] , ' is a match.'
        matchtime = matchtime + time.time()
    fh.close
matchtimelist.append(matchtime)
progtimelist.append(time.time() - progstart)
print comment,"\n", float(sum(progtimelist))/len(progtimelist) , 
max(matchtimelist)
f = open( ofile, 'a', 0)
sampled = int((sstop-sstart+1)/sint)
f.write("\n\n" + str(datetime.datetime.now()) + "n" + comment + 
"n" + str(t) + " iterations\n" + "m=" + str(m) + " k=" + str(k) + "\nfiles=" + ifile + "\nstart = " + str(sstart) + " stop = " + str(sstop)
+ " sint = " + str(sint) + " sampled = " + str(sampled) + "\nProg: Av time= " + str(round(float(sum(progtimelist))/len(progtimelist)),2))" + "s, Min time= " + str(round(min(progtimelist),2)) + "s, Max time= " + str(round(max(progtimelist),2))" + "s, Range= " + str(round(max(progtimelist) - min(progtimelist),2)) + "s\nMatch: Av time= " + str(round(float(sum(matchtimelist))/len(matchtimelist)),2)) + "s, Min time= " + str(round(min(matchtimelist),2)) + "s, Max time= " + str(round(max(matchtimelist),2)) + "s, Range= " + str(round(max(matchtimelist) - min(matchtimelist),2)) + "s\n\n"
f.close
#module: usebf.py
#description: read sectors and match hashes using Bloom filter. Record results.
#author: John Clayton, MSc dissertation
#last amended: 15 August 2012
import os, time, datetime, random

def usebfm(comment,t,sint,m,k,iimage):
    '''Function to run Bloom filter match against data image and write results to csv file'''
    frag = int(iimage[-4])
    from datetime import datetime
    import sqlite3 as lite
    from bitarray import bitarray
    progtimelist, disktimelist, matchtimelist, tposlist, fposislist, fposbflist,
    accuracylist = [], [], [], [], [], []
    # Bf m is size in bits (must be multiple of 4 , b is conversion to bytes, k is
    # number of hash functions
    b = int(m/float(4))
    #use one of available Bloom filters
    bloomfile = "bloom" + str(m) + "-" + str(k) + ".bin"
    # image file, sstart is start sector, sstop is stop sector (+1 as it will define
    # a range), sint is interval
    # these all specific to IMG_0036.JPG in fragmented files not generic
    # ifile image file location
    ipath = "/home/john/Documents/MScDissertation/images/"
    if ifile == "256Mjcusbf1.dd":
        vstart0, vstop0, vstart1, vstop1, vstart2, vstop2, sstop =
        57464,59187,57464,59187,57464,59187,489471
    elif ifile == "256Mjcusbf2.dd":
        vstart0, vstop0, vstart1, vstop1, vstart2, vstop2, sstop =
        57464,58612,57464,58612,158613,159187,489471
    elif ifile == "256Mjcusbf3.dd":
        vstart0, vstop0, vstart1, vstop1, vstart2, vstop2, sstop =
        57464,58037,158613,159187,258038,258612,489471
    sampled = 0
    # xtpos is expected number of hits for sector that size limited to 1 as its a
    # probability
    xtpos1724 = float(1724)/sint
    if xtpos1724 > 1:
        xtpos1724 = 1
    xtpos1149 = float(1149)/sint
    if xtpos1149 > 1:
        xtpos1149 = 1
    xtpos574 = float(574)/sint
    if xtpos574 > 1:
        xtpos574 = 1
    xtpos575 = float(575)/sint
    if xtpos575 > 1:
        xtpos575 = 1
    # xntposrate is expected rate of no hits based on frags
    if frag == 1:
        xntposrate = 1 - xtpos1724
    else:
        xntposrate = 1 - xtpos1724


elif frag == 2:
xntposrate = (1-xtpos1)*(1-xtpos5)
elif frag == 3:
xntposrate = (1-xtpos5)*(1-xtpos5)*(1-xtpos4)
else:
xntposrate = 99
# lists to record for each iteration
for j in range(t):
sstart = random.randint(0,sint-1)
tpos, fposis, fposbf, matchtime, disktime = 0, 0, 0, 0
progstart = time.time()
b = bitarray()
with open(bloomfile,'rb') as fh:
b.fromfile(fh)
for s in range(sstart,sstop,sint):
match = 0
c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
disktime = disktime - time.time()
os.system(c)
result=os.popen(c).read()
disktime = disktime + time.time()
sampled +=1
if match == 1:
if  (vstart0 <= s <= vstop0) or (vstart1 <= s <= vstop1) or (vstart2 <= s <= vstop2):
tpos += 1
print j, ' Sector ', s, ' hash ', result[:32] , ' is a true +ve match'
else:
con = lite.connect('hashes.db')
d="Select * FROM Hashes WHERE Hash='%s'
result=os.popen(c).read()
disktime = disktime + time.time()
data=cur.fetchone()
accuracylist.append(1)
# xtposraterand is expected rate of no hits based on random looking for 1 off
1718 distinct sectors
xntposraterand = 1
for ii in range(1,sampled+1):
    xntposraterand = xntposraterand*(sstop-(i-1)-1718)/float(sstop-(i-1))
    xtposraterand = (1 - xntposraterand)

f = open("results.txt", 'a', 0)
f.write("\n\n" + str(datetime.now()) + "\n" + comment + "\n" + str(t) + " iterations; " + str(sampled) + " total sectors sampled\n" + "m=" + str(m) + " k=" + str(k) + " niimage=" + iimage + "; frag =" + str(frag) + "\n\nsstot = " + str(sstop) + "\n" + str(sint) + " Avg time =" + str(round(avgprogtime,2)) + "\n" + str(round(min(progtimelist),2)) + " Max time =" + str(round(max(progtimelist),2)) + "\nAvg Prog: Av time =" + str(round(averageprogtimes,4)) + " Min time =" + str(round(min(progtimelist),4)) + " Max time =" + str(round(max(progtimelist),4)) + " Range = " + str(round(max(progtimelist) - min(progtimelist),4)) + " SD = " + str(round(progtimesd,4)) + "\nMatch: Av time = " + str(round(averagematchtimes,4)) + " Min time = " + str(round(min(matchtimelist),4)) + " Max time = " + str(round(max(matchtimelist),4)) + " Range = " + str(round(max(matchtimelist) - min(matchtimelist),4)) + " SD = " + str(round(matchtimesd,4)) + "\nDisk: Av time = " + str(round(averagedisktimes,4)) + " Min time = " + str(round(min(disktimelist),4)) + " Max time = " + str(round(max(disktimelist),4)) + " Range = " + str(round(max(disktimelist) - min(disktimelist),4)) + " SD = " + str(round(disktimesd,4)) + "\nAv True+ve = " + str(sum(tposlist)/len(tposlist)) + " Av False+ve(is) = " + str(sum(fposislist)/len(fposislist)) + " Av False+ve(bf) = " + str(sum(fposbflist)/len(fposbflist)) + " Av Accuracy = " + str(float(sum(accuracylist))/len(accuracylist)) + "\nTot True+ve = " + str(sum(tposlist)) + " Tot False+ve(is) = " + str(sum(fposislist)) + " Tot False+ve(bf) = " + str(sum(fposbflist)) + "\nExpected True+ve Hit checkrate frag = " + str(round((1-xntposrate),4)) + " Expected True+ve Hit checkrate rand = " + str(round((xtposraterand),4)) + " Actual True+ve Hit checkrate = " + str(round(1-(float(tposlist.count(0))/len(tposlist)),4)) + " Number of True+ve Hit checks = " + str((t-tposlist.count(0))) + "\nAv throughput = " + str(round(throughput,4)) + " kB/s ; Av scanrate = " + str(round(scanrate,4)) + " MB/s\n\n"
f.close
#comment,t,sint,m,k,ifile,frag
comment="Run as function using crontab"
t=1
for iimage in ("30Mnpssf1.dd","30Mnpssf2.dd","30Mnpssf3.dd"):
    for m in (16,12):
for k in (8, 4):
    for sint in (500, 1000, 2000, 4000):
        usebfm(comment, t, sint, m, k, iimage)