

A Complex Situation in Data Recovery

A Thesis submitted for the degree of Doctor of Philosophy

By

J. Sue Ashworth

**School of Information Systems, Computing and Mathematics
Brunel University**

May 2009

Acknowledgements

Special thanks to Martin for being my boyfriend and to every one who reminded me that I should be nice to my boyfriend.

Abstract

The research considers an unusual situation in data recovery. Data recovery is the process of recovering data from recording media that is not accessible by normal means. Providing that the data has not been overwritten or the recording medium physically damaged, this is usually a relatively simple process of either repairing the file system so that the file(s) may be accessed as usual or finding the data on the medium and copying it directly from the medium into normal file(s). The data in this recovery situation is recorded by specialist call centre recording equipment and is stored on the recording medium in a proprietary format whereby simultaneous conversations are multiplexed together and can only be accessed by using associated metadata records. The value of the recorded data may be very high especially in the financial sector where it may be considered a legal audit of business transactions. When a failure occurs and data needs to be recovered, both the data and metadata information must be recreated before a single call can be replayed. A key component to accessing this information is the location metadata that identifies the location of the required components on the medium. If the metadata is corrupted, incomplete or wrong then a repair cannot proceed until it is corrected.

This research focuses on the problem of verifying this location metadata. Initially it was believed that only a small set of errors would exist and work centred on detecting these errors by presenting the information to engineers in an at-a-glance image. When the extent of the possible errors was realised, an attempt was made to deduce location metadata by exploring the content of the recorded medium. Although successful in one instance, the process was not able to distinguish between current and previous uses. Eventually insights gained from exploration of the recording application's source code, permitted an intelligent trial and error process which deduced the underlying medium apportioning formula. It was then possible to incorporate this formula into the heuristics, generating the at-a-glance image, to create an artefact that could verify the location metadata for any given repair.

After discovering the formula, the research returned to the media exploration and the produced disk fingerprinting technique. The disk fingerprinting technique gave valuable insights into error states in call centre recording and provided a new way of seeing the contents of a hard drive.

This research provided the following contributions:

- 1. It has provided a means by which the recording systems' location metadata can be verified and repaired.*
- 2. As a result of this verification, greater automation of the recovery process is now possible before the need for human verification is required.*
- 3. The disk fingerprinting process. This has already given insights into the recording system's problems and is able to provide a new way of seeing the contents of recording media.*

Table of Contents

Glossary.....	9
Abbreviations and Acronyms.....	9
Unix Commands.....	9
RAID – Quick Overview.....	13
1 Introduction.....	15
1.1 This research.....	15
1.2 Context of the Problem – Telephone Voice Recording.....	16
1.3 The Problem	19
1.4 This Situation - Context and Research Environment.....	22
1.5 Objectives.....	22
1.6 Structure of Thesis.....	23
2 Background of this Research.....	26
2.1 Introduction.....	26
2.2 A Typical PC: Functioning and Data Storage.....	26
2.3 The Recording System: Functioning and Data Storage.....	30
2.4 When Something Goes Wrong: The Operating System.....	34
2.5 When something Goes Wrong: the Data Partition.....	37
2.6 Causes of Failure.....	43
2.7 Company Work Practices and Data Recovery Future Plans.....	45
2.8 Disk fingerprinting.....	47
2.9 Chapter Summary	51
3 Visualising the data structures.....	53
3.1 Background	53
3.2 Introduction.....	54
3.3 The abslsr.txt	54
3.4 Design Decisions.....	56
3.5 First Iteration.....	57
3.6 Non Linear Representations.....	59
3.7 The Most Pertinent Information.....	60
3.8 Added Value.....	62
3.9 Conclusions and Lessons Learnt.....	65
4 Disk Exploration and Fingerprinting.....	67
4.1 Background.....	67
4.2 Introduction.....	68
4.3 Missing Information.....	68
4.4 Practical Considerations.....	69
4.5 Searching the Disk.....	70
4.6 Putting the Theoretical Values to the Test.....	79
4.7 Wider Considerations.....	80
4.8 Disk Fingerprinting.....	81
4.9 Chapter Conclusions.....	85
5 Calculating the Correct File Sizes.....	87
5.1 Background.....	87
5.2 Introduction.....	87
5.3 The Code.....	88

5.4 The Files and their Calculations.....	89
5.5 Testing.....	92
5.6 Using the Information – Design and Implementation.....	92
5.7 Using the Information as Part of the Repair Process.....	94
5.8 An Example Beyond Repair.....	95
5.9 One Final Consideration.....	99
5.10 Chapter Summary.....	102
6 Disk Fingerprinting – An Exploration.....	104
6.1 Background.....	104
6.2 Introduction.....	104
6.3 Common Methods.....	105
6.4 Disk 1 General.....	106
6.5 Comparing Two C: Drives.....	115
6.6 Formatting a FAT 16 Drive.....	118
6.7 A Linux Disk.....	121
6.8 Two Fresh Installs of Linux.....	124
6.9 Similarities in File Storage.....	132
6.10 Disk Fingerprinting Round Up.....	137
7 Summary and Conclusions.....	140
7.1 Introduction.....	140
7.2 Summary.....	140
7.3 Achievements.....	142
7.4 Enabled and Future work.....	143
7.5 Contributions.....	144
Appendices.....	146
Appendix A –Data Storage.....	146
Appendix B – Description and Source Code of Sample Puzzle.....	146
Appendix C – Source Code for SVG Generation.....	149
Appendix D – Source Code for Disk Iterating Script.....	156
Appendix E – Source code for Web Server Calculator.....	156
References.....	170

List of Figures

Figure 1.2.1: A typical recording system.....	17
Figure 1.6.1: A diagram of the research processes	24
Figure 2.2.1: A sample boot sector.....	28
Figure 2.5.1: A typical history.....	39
Figure 2.5.2: A repaired file.....	39
Figure 2.5.3: A misaligned file.....	41
Figure 2.8.1: gnuplot basic.....	49
Figure 2.8.2: Linux laptop fingerprint.....	50
Figure 2.8.3: Start of a typical data partition	50
Figure 3.3.1: A typical abslsr.txt.....	55
Figure 3.5.1: Initial unscaled graphic.....	57
Figure 3.5.2: Partially zoomed.....	58
Figure 3.5.3: Fully zoomed.....	58
Figure 3.5.4: Single axis zoom.....	58
Figure 3.6.1: Log base - e.....	60
Figure 3.6.2: Log base - 10.....	60
Figure 3.7.1: The number of records unzoomed.....	61
Figure 3.7.2: The number of records zoomed.....	61
Figure 3.7.3: Number of records in log base-10.....	62
Figure 3.8.1: Grouping files.....	63
Figure 3.8.2: Adding intelligence.....	64
Figure 3.8.3: The discrepancy.....	65
Figure 4.8.1: Raw graph.....	82
Figure 4.8.2: Annotated graph.....	83
Figure 4.8.3: Second disk - raw.....	84
Figure 4.8.4: Second disk - annotated.....	84
Figure 5.4.1: Output from sg_readcap.....	90
Figure 5.6.1: Data input web page.....	93
Figure 5.6.2: Metadata predicted layout.....	94
Figure 5.8.1: The start of the partition where the metadata should be.....	96
Figure 5.8.2: A portion of a normal disk.dat.....	98
Figure 5.9.1: Data input form.....	99
Figure 5.9.2: A good abslsr.txt file.....	100
Figure 5.9.3: The corrected layout with abslsr.txt errors depicted in red below.....	101
Figure 6.4.1: A screen shot of Partition Magic.....	109
Figure 6.4.2: First eighth of drive.....	110
Figure 6.4.3: Second eighth of drive.....	111
Figure 6.4.4: Third eighth of drive.....	111
Figure 6.4.5: Forth eighth of drive.....	112
Figure 6.4.6: Fifth eighth of drive.....	112
Figure 6.4.7: Sixth eighth of drive.....	113
Figure 6.4.8: Seventh eighth of drive.....	113
Figure 6.4.9: Final eighth of drive.....	113
Figure 6.5.1: C:1.....	116
Figure 6.5.2: C:2.....	116
Figure 6.5.3: C:1 start.....	117

Figure 6.5.4: C:2 start..... 117

Figure 6.6.1: C:1 before format..... 119

Figure 6.6.2: C:1 after format..... 119

Figure 6.6.3: C:1 pre format..... 120

Figure 6.6.4: C:1 post format..... 120

Figure 6.7.1: Boot partition..... 123

Figure 6.7.2: Some typical patterns..... 123

Figure 6.8.1: Boot partition and start of LVM unencrypted..... 126

Figure 6.8.2: Boot partition and start of encrypted LVM..... 126

Figure 6.8.3: Unencrypted LVM part 2..... 127

Figure 6.8.4: Unencrypted LVM part 3..... 127

Figure 6.8.5: Unencrypted LVM part 4..... 127

Figure 6.8.6: Unencrypted LVM part 5..... 128

Figure 6.8.7: Unencrypted LVM part 6..... 128

Figure 6.8.8: Unencrypted LVM part 7..... 128

Figure 6.8.9: Encrypted LVM part 2..... 129

Figure 6.8.10: Encrypted LVM part 3..... 129

Figure 6.8.11: Encrypted LVM part 4..... 130

Figure 6.8.12: Encrypted LVM part 5..... 130

Figure 6.8.13: Encrypted LVM part 6..... 130

Figure 6.8.14: Encrypted LVM part 7..... 130

Figure 6.9.1: /boot partition and start of the LVM..... 132

Figure 6.9.2: Part 2 of the LVM..... 133

Figure 6.9.3: Part 3 of the LVM..... 133

Figure 6.9.4: Part 4 of the LVM..... 133

Figure 6.9.5: Part 5 of the LVM..... 133

Figure 6.9.6: Part 6 of the LVM..... 134

Figure 6.9.7: The end of the LVM and the start of the NTFS partition..... 134

Figure 6.9.8: Part 2 of the LVM after text files copied..... 135

Figure 6.9.9: Part 3 of the LVM after text files copied..... 135

Figure 6.9.10: Fingerprint text data on part 3 of the LVM 136

Figure 6.9.11: Drive I: containing the data from scanning of C: drives..... 137

Glossary

The glossary is to explain the technical terminology that is endemic in data recovery, Unix, and this research. This guide is intended to refresh the expert and provide a guide for the less technically oriented. This glossary covers abbreviations and acronyms; Unix commands and a brief overview of types of RAID (Redundant Array of Independent Drives).

Abbreviations and Acronyms

CD	Compact Disk
DOS	Disk Operating System
DVD	Digital Versatile Disk
ext3	EXTended file system type 3. A file system used by Linux.
FAT	File Access Table
GUI	Graphical User Interface
LVM	Logical Volume Manager, a Linux file system management system
NTFS	New Technologies File System, a Microsoft proprietary file system
PC	Personal Computer
PNG	Portable Network graphics, an image format
RAID	Random Array of Independent Drives
SCSI	Small Computer System Interface
SVG	Scalable Vector Graphics, an image format
UDF	User Defined Field. A field that contains information chosen by the user.
USB	Universal Serial Bus
VPN	Virtual Private Network

Unix Commands

This section contains a brief overview of the commands used in this research. For a full

description of any command the Unix man (manual) pages are the definitive source of information. These are accessed from the command line using the command “man command” (where command is the command about which more information is required). For more information about the man pages typing “man man” will access the manual for man. The manual pages typically contain the NAME of the command, a SYNOPSIS, a DESCRIPTION, the name of the AUTHOR, a REPORTING BUGS address, COPYRIGHT information and a SEE ALSO section. Different man pages may contain other, additional headings but these are the most usual.

Most of the following descriptions are based on the commands' man pages.

|

The pipe symbol means to take the output of the command to the left of the symbol and feed it into the command on the right, e.g. `cat this.txt | less` will take the output of the `cat` command and feed it to `less` where the text will be displayed.

awk

This is a pattern scanning and processing language that is designed for manipulating text data in files or piped from another program.

cd

The change directory command is used to move from one directory to another when working at the command line.

cat

Cat is short for concatenate. This is typically used with files as the input and the console as the output.

dd

This is a convert and copy utility. This command reads data from a file or block device and outputs the data to another file or block device. The input is specified by `if`, e.g. `if=/dev/sda` will set the input to read from the first SCSI drive. The output is specified by `of`, e.g. `of=/dev/sdb` will set the output to the second SCSI drive. If no

output is given then the default is `stdout` (the console). The block size is set with the `bs` assignment, e.g. `bs=512`. The number of blocks to be copied is set by the `count` value, e.g. `count=100`. The `skip` value is the offset into the input file to start copying, while the `seek` value is the offset into the output to start writing. For example: `dd if=/dev/sda of=my.part.img bs=512 count=1` will copy the first sector of the first SCSI disk to a file. This would be a reasonable way to make a backup of the partition table.

`diff`

This finds the differences between two or more files.

`dosfsck`

This is a Linux utility that can perform scandisk like functions on a FAT file system.

`fdisk`

Fdisk is a partition table manipulator for Linux and other systems. Used with the `-l` switch it displays the current partitions.

`grep`

This prints lines matching a pattern. That is to say it only shows the lines that match a given pattern.

`hexdump`

Hexdump is an ascii, decimal, hexadecimal or octal dump. The hexdump utility which displays specified files in a user decided format. When used in this project the following formats were chosen: The canonical hex+ASCII display which displays the input offset in hexadecimal, followed by sixteen space-separated, two column, hexadecimal bytes, followed by the same sixteen bytes in `%_p` format enclosed in “|” characters and the two-byte decimal display which displays the input, offset in hexadecimal, followed by eight space-separated, five column, zero-filled, two-byte units of input data, in unsigned decimal, per line.

`less`

Less is a program similar to more, a text editor, but which allows backward movement in the file as well as forward movement. Less does not have to read the entire file before it can display information so, with large input files, it starts up faster than text editors such as vi.

`sed`

Sed is a Stream EDitor for filtering and transforming text. A stream editor can be used to perform basic text transformations on an input stream (possibly a file or input from a pipe). Sed's ability to filter text in a pipeline particularly sets it apart from other editors.

`sg_readcap`

This utility is not normally part of the Unix operating system and can be found at http://sg.danny.cz/sg/uu_index.htm.

This command sends a SCSI READ CAPACITY command. The SCSI READ CAPACITY command actually yields the block address of the last block and the block size. The number of blocks is thus one plus the block address of the last block (as blocks are counted origin zero).

`ssh`

Ssh (SSH client) is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to replace rlogin and rsh, and provide secure encrypted communications between two untrusted hosts over an insecure network.

`su`

SU is short for super user. It is used to gain elevated privileges.

`uniq`

uniq program reports or omits repeated lines. It discards all but one of successive identical lines from INPUT (or standard input), writing to OUTPUT (or standard output). As used in this research with the -c switch meaning to prefix the output with a count of the number of occurrences.

vi

Vi is a text editor. It can be used to edit all kinds of plain text.

wc

Wc stands for word count and it is used to count words, lines or bytes.

RAID – Quick Overview

This brief explanation of RAID provides a background to RAID systems and identifies the two types encountered in the project. The acronym RAID originally stood for “Redundant Array of Inexpensive Disks”, but as “inexpensive” was somewhat vague and relative, the “I” now stands for “Independent”.

Although software RAIDs can be created simulating some of the advantages of multiple disks, the term usually applies to more than one physical hard drive combined to form an array of drives. All references to RAID in this research refer to more than one physical hard drive.

There are several types of RAID numbered 0 to 6, each level providing redundancy, improved performance or a combination.

RAID 0: This configuration provides striping (a means of distributing data across multiple drives so that usually disk I/O bottlenecks can be minimised) and does not use parity, or any other error detection, to provide fault tolerance. The more drives present in the array the more vulnerable to failure the array becomes. This RAID may be used to increase performance.

RAID 1: This is a mirrored set of disks. Mirroring occurs where the data on one disk is replicated on another. With this configuration the data is recoverable as long as one disk is sound. This is one configuration used by the voice recording system, described in this research, where two mirrored disks are fitted internally.

RAID 2: Here the disks are striped, but with very small stripes and with the disks synchronised to spin together. Due to the use of error correction this configuration can recover from single bit corruption and detect, but not repair, double bit. RAID 2 is very poor in terms of storage. Within a 7 disk array four disks are used for data storage and the remaining three for error correction.

RAID 3: In this configuration byte sized stripes are used and data is spread sequentially across several disks with another disk used purely for parity. This configuration does not permit multiple I/O requests simultaneously as each block read entails all drives being read sequentially and unable to seek to another block at the same time.

RAID 4: This configuration uses block size striping and has one disk dedicated to parity. Unlike RAID 3 if the read (the amount of data read at a time) is smaller than a block then another disk may read a different block if this is located on a different drive.

RAID 5: This is similar to RAID 4 except that here the parity is distributed between the disks. This configuration means that if one drive fails there is no loss of data and if a new drive is installed the RAID can rebuild itself. Typically a RAID enclosure may contain several drives plus a dedicated spare. When one drive fails the spare is used in its place and the data from the failed drive is built from the data and parity information stored across the remaining drives. In this research this type of RAID is used if an external RAID is present in the voice recording system.

RAID 6: This is similar to RAID 5, but it has two sets of distributed parity thus at any given moment in time there may be two failed drives without loss of data or, and possibly more commonly, when a second drive fails while the first is being rebuilt. This gives a high degree of redundancy. Like RAID 4 and 5 it uses block level striping.

In this research RAID 1 and 5 were encountered. In general use of RAIDs 0, 1 and 5 are common, while the remaining levels are seldom encountered.

1 Introduction

1.1 This research

Data recovery is the process by which data that was written to recordable media and has become inaccessible is made available once more. The data may be of any type from a single plain text document to a full operating system with all a user's documents or a proprietary structure used for a specific purpose. The recordable storage media may be a hard drive, a floppy disk, a CD or DVD, a (digital) tape, a USB flash memory device, a RAID system or any other media that is able to store computer data. The cause of the data being inaccessible may be due to physical damage to the media or corruption of the file system, partition table or other metadata used to access the data. The nature of the recovery process will depend on the underlying reason for the data being inaccessible.

This research is about data recovery in the field of call centre telephone voice recording. It is about a specific method of audio recording and the implications of this recording technique when failures occur and the data needs specialist attention to recover the stored information.

Call centres may be extremely busy with a large number of calls occurring simultaneously. In this system these calls are multiplexed (combined into a single data stream) while the information needed to demultiplex the data is stored in separate metadata files. There is no underlying file system and the different metadata streams are recorded to discrete areas of the recording medium, normally a hard disk or a Random Array of Independent Drives (RAID) system. The locations of these metadata files are defined in a location metadata block and are determined by a software formula that takes into consideration the recording equipment's hardware and customer refinements. This is a recording strategy that is normally very reliable, but when things do go wrong the result is extremely unpredictable.

The first step in the recovery process is to examine the metadata information at the location indicated by the location metadata. If this location information is wrong then the repair can not

proceed until it is corrected. For this system, the underlying formula by which the medium is apportioned has been lost. This means that where this data is corrupted, incorrect or missing there may be no reliable way to detect these errors or determine the correct information if the problem is identified. The focus of this research is the recreation of the formula by which the metadata locations are determined.

During the research process an attempt to reconstruct location metadata by exploring the contents of a disk produced a potentially very useful diagnostic technique. The technique of disk fingerprinting has provided very useful insights to data recovery and in a more general context.

The work was done in the laboratory of the company owning the voice producing the call centre recording equipment, this company shall be only referred to as company X and the recording product as recorder X or more simply as the recorder.

1.2 Context of the Problem – Telephone Voice Recording

When initiating a business telephone conversation it is not unusual to hear some variation of the phrase “Calls may be recorded for training purposes”. While a reason for informing a caller of the recording is to allow them to cancel the call if being recorded is unacceptable, it also provides notice of the company's intentions. A company may legally record conversations with its customers and potential customers and retain the data for later use.

The training purposes alluded to suggest exemplary calls being played to new recruits to train them in the company's best practices, while some calls may be used in this way, there is considerably more useful information in the recorded data. Many of these business calls will be handled by a call or contact centre. (Typically a call centre will only communicate via telephone while a contact centre may also use fax or email etc. to supplement the communications.) The employees, or agents, based in such a centre will typically each have a workstation comprising of a PC and a telephone headset (with which to make or take calls).

Figure 1.2.1 shows a typical overview of a call centre recording system. Here it can be seen that the company's Private Branch eXchange (PBX) is connected to the telecommunications supplier's Public Switched Telephone Network (PSTN). The agents' conversations travel through the company's BPX and are recorded by the recording system. The telephone recorder system may contain hard drive and tape data storage. It is this storage media with which a data recovery engineer will typically work.

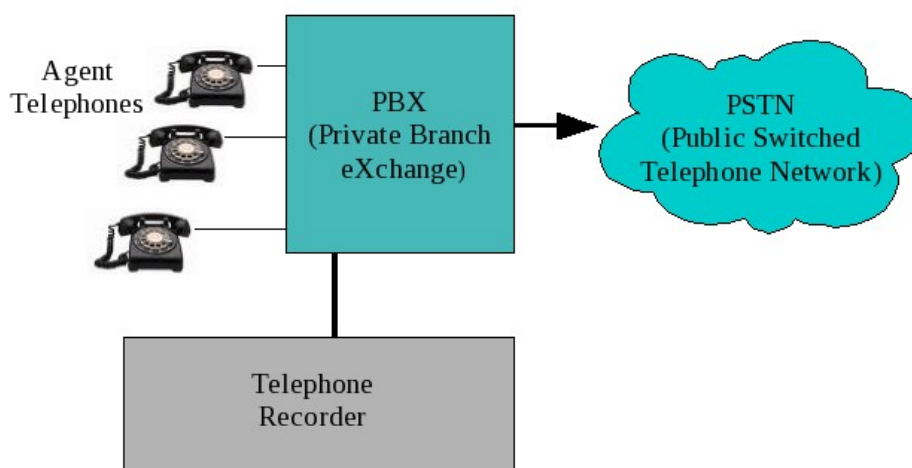


Figure 1.2.1: A typical recording system

These centres are used for many different purposes, a centre that focuses on telemarketing will have employees making or following up calls with a view to making a sale. A key performance indicator (KPI) for this scenario is the number of sales per given time interval. The centre may be run for a bank providing a telephone banking service or it may be a centre for car insurance claims, In these circumstances a KPI could be the number of calls handled per hour. Different companies will set differing targets and different business objectives for their centres, but most will seek to optimise the performance of their employees.

Optimisation may start by a customer being asked to key (or speak) a number or sequence of numbers to select an option before being connected to the best-suited agent. An agent will be closely monitored, the amount of time spent talking to any individual customer is recorded as is the number of calls per hour, the interval between calls and the amount of time an agent is unavailable when taking a break. With this type of information the performance of an agent can be examined and if required it may be improved, for example, by additional training to shorten

the time spent talking to a customer while still retaining a satisfactory outcome. Agents' performance can be compared with that of other agents and the results used can identify effective traits, as can follow-up customer surveys. Some of the insights gained from the recorded calls may be gleaned from voice recognition software. Performance-enhancing software may detect when an agent requires further training and launch a PC based training session at their workstation. The description "used for training purposes" can mean that all customer calls are analysed and their content data mined so that more efficient strategies can be taught to call centre employees. This data is potentially very useful to its owners.

Occasionally disputes may arise, a customer may believe they said one thing whilst the company may believe it received differing instructions, in such cases replaying the recording of the conversation will quickly and easily resolve the issue. More complex situations may occur in which a recording may be required as evidence for a court of law, e.g. where funds are being moved in as part of an illegal operation or where the caller is not whom they claim to be. The precise nature of such situations is beyond the scope of this project. It should be noted that that if a recording is used as legal evidence, a degree of authenticity will be required. If data recovery has been performed then some assurance must be presented that the recovered data is the same as the original recording. There is no reliable method of predicting which calls will be deemed valuable and where any call may be potentially valuable then all calls must be considered valuable. When performing data recovery on a recording it should be remembered that a nondescript repair may subsequently become important audio evidence that must be demonstrated, to a court, to be a realistic representation of a conversation between the parties that took place on the date and time in question. Fortunately such occurrences are extremely rare but the possibility is always present.

Although there is profit in being able to refine the workforce to a high degree, there is also value in retaining recorded calls as an entity in their own right and under some circumstances a level of recording may be mandatory. Consider the case of a bank customer telephoning their bank and transferring money between two accounts, a normal everyday occurrence, but one that only a few years ago would have required the customer to visit their bank in person and give the cashier a signed transfer document. With the advent of telephone banking (and other similar services) the recorded transcript is now the authenticated audit trail replacing the transaction slip. Financial

institutions have regulators and just as the loss of signed transfer documents, cheques or other evidence of authenticated transaction is not acceptable neither is the loss of recorded data. Financial regulators appreciate that 100% data integrity, though desirable, may not be achievable and apply a less stringent target of 99.XX % of data being available. The exact percentage may vary with the type of service provided, but there is a real risk that an organisation could lose its licence to operate if too much data is lost. Here the value of the volume of calls may be disproportionately high compared to the sum of the values of the individual calls.

Modern telephone recording systems are reliable and to lose a conversation is rare. Once a call is initiated the audio data is recorded to storage media and periodically this data will be backed up to longer-term storage media, such as magnetic tape, and archived. As with any data, the most vulnerable period is after it has been created but before a reliable backup has been made.

1.3 The Problem

The biggest aspect of the problem is the reliability of the system and the lack of common faults. This means that although there are some faults that have precedence almost every data recovery job is a new challenge. An internal company document likens the analysis of a repair to that of an air accident investigation where all the fragments from a downed aircraft must be found, their original location determined and an analysis of the cause of the accident undertaken. When an aeronautical disaster occurs there may be speculation to the sequence of events, but until the investigation is completed the root cause is unknown. Like the work of air accident investigators, each repair starts with a search for all the pieces of the data and the correct location determined. Here, unlike an air accident investigation, after the analysis the data must be reassembled and made playable again.

With very few exceptions it is not possible to observe symptom X and deduce that approach Y is the correct method to follow. Even if the underlying cause is suspected there is no certainty that the effect will be the same as a previous occurrence. Another consideration is that the time at which a failure is recognised may not be the same as the time of failure. Sometimes there is an

obvious cause for the failure, for example after a power outage no data could be replayed. Sometimes a more insidious sequence of events may occur and the first recognised symptom is a failure in the backup process. If a problem is recognised immediately then the recording medium will be taken out of service, but it is possible that recording may continue and new data will be written to the hard drive or RAID system. This may mean that some data is overwritten but more often it is just another factor in an already complex situation.

The causes of failure may be hardware, software or ill considered human intervention. For example hard drives may fail (although RAID systems can provide data protection in the event of a single disk failure). Software problems can also jeopardise recorded data and there are wide ranges of possible causes ranging from software bugs that result in an application writing to the wrong part of the disk to configuration issues erroneously resetting significant parameters. Sometimes a problem may be compounded rather than alleviated during the initial system fault assessment.

Irrespective of history, all repairs must begin with an examination of the data and its associated metadata, but in order to access these files their location must be known. The location of the data and metadata components is stored in the location metadata and it is the contents of this file that permits the collection and examination of the information required for a repair to proceed. If the location information is wrong, incorrect portions of the medium will be examined leading to an incomplete or missing capture of the data. The location of these files is determined by a software formula when the recording system is commissioned. Without knowing the formula used to apportion the recording medium the following problems exist:

1. If the location metadata was corrupted or modified inadvertently then there is no way to deduce the correct layout nor (and this may ultimately be more significant) is there any way to validate if the information present is correct.
2. Where the location metadata is changed the problem may be twofold, first there may be data recorded prior to the change that has become inaccessible and second there may be subsequent recordings written to the wrong place. The correct location metadata must be deduced before calls recorded prior to the incident can be accessed with the same

information required for recordings made after the incident to be moved to the correct location on the recording media.

3. If the location metadata is missing there is very little to assist the recovery engineer to begin the recovery process.

Fortunately the location metadata is usually correct and even if the information can not be verified it is possible to proceed on the basis that it is unlikely to be wrong. For the majority of repairs this is a strategy that works. Where this strategy is employed the recovery engineer will usually seek supporting information as the repair proceeds (e.g. the metadata files contain expected data) and by the time the assessment is complete there is a high degree of confidence in the information. However if errors exist and go undetected then a good deal of time may be lost in backtracking and understanding why a repair will not replay. Data recovery engineers' time is an expensive commodity to waste in this fashion.

Although the above verification is performed by considering if other information is consistent with the collected location metadata. It would be advantageous for the location information to be otherwise verified and then used to confirm the supporting data, rather than observing consistency (which may or may not be indicative of correctness).

Although the immediate user of the location metadata is the data recovery engineer, there is a future plan to create a Recovery Studio, which will permit a less experienced engineer, possibly the front line maintenance engineer, to perform some basic recovery repairs or assessments. As well as considering the nature of the recording medium allocation formula, some additional consideration must be given to the presentation of the information both to an experienced user and the novice. The Recovery Studio concept is considered in greater detail in section 2.7

1.4 This Situation - Context and Research Environment

Most of the experiments described in this research took place while employed three days a week

as a data recovery engineer. Although this employment did not impact directly on the experiments performed nor the choices of the approaches undertaken, it is worth briefly considering how it influenced the research.

This research took place in the company's data recovery laboratory while employed by the company. The work was carried out under controlled conditions and these were the normal conditions for this type of data recovery. This meant that distinctions of *in vivo* or *in vitro* (Basili 2007) were not applicable. As the research was carried out while working as a data recovery engineer, this permitted both aspects of the practitioner and researcher approaches to be experienced (Basili 1996). These advantages of being able to research in this manner were complimented by a degree of sensitivity to the company's requirements.

As this research mostly took place while employed as a data recovery engineer there were occasions where company core business was expected to take precedence over personal research considerations. This was not an unreasonable expectation. The company's expectations were that during normal working hours the focus of attention should be company business. Outside these hours there was the freedom to use data for research purposes. A remote login was arranged for this purpose and to permit some company work to be performed from home.

The employment situation lasted for a year before economic circumstances forced the company to reduce staff and as a recent employee and part time worker, the employment concluded. By this time the research had successfully identified disk allocation formula.

1.5 Objectives

The overall aim of this research is to improve the the initial analysis of the data recovery repair process. To do this, the primary objectives for this research are:

1. To attempt to identify the formula used in apportioning the recording medium into areas of metadata and data.

2. If found, to present this information in a usable form for both expert and novice users so that they may make an informed initial assessment of the repair.
3. To see what, if any, useful information could be obtained from the disk fingerprinting technique developed as an attempt to automate the exploration of recording medium (see chapter 2 section 8).

1.6 Structure of Thesis

The rest of the thesis is organised as follows:

Chapter 2 considers the nature of an operating system and the types of problems that can arise when key parts of the recording medium are damaged. These principles are extended to the call centre recording system and the application specific data recording mechanism is explored. The problems that may be caused by this recording stratagem are also considered. The main causes of failure are identified before considering the current working practices and the company's long term aspirations. A description of the disk fingerprinting technique is also included here before a brief summary of the chapter's main concepts.

The following four chapters begin with a brief consideration of the external factors present when the work was undertaken. Chapters 3 to 5 describe the different stages of solving the problem of verifying and recreating the location metadata. Figure 1.6.1 shows a flow diagram of the work done in these stages.

Chapter 3 describes the process of extracting information from the location metadata; performing some heuristic analysis and presenting this information visually to the user with any inconsistencies highlighted (an at-a-glance representation). While the objectives of the processes were fully met, the initial understanding of the problem was incomplete and subsequent repair jobs revealed that key data may not only be incorrect but also missing. Nothing in the processes

developed in this chapter could cope with missing information. This is depicted in the top third of Figure 1.6.1.

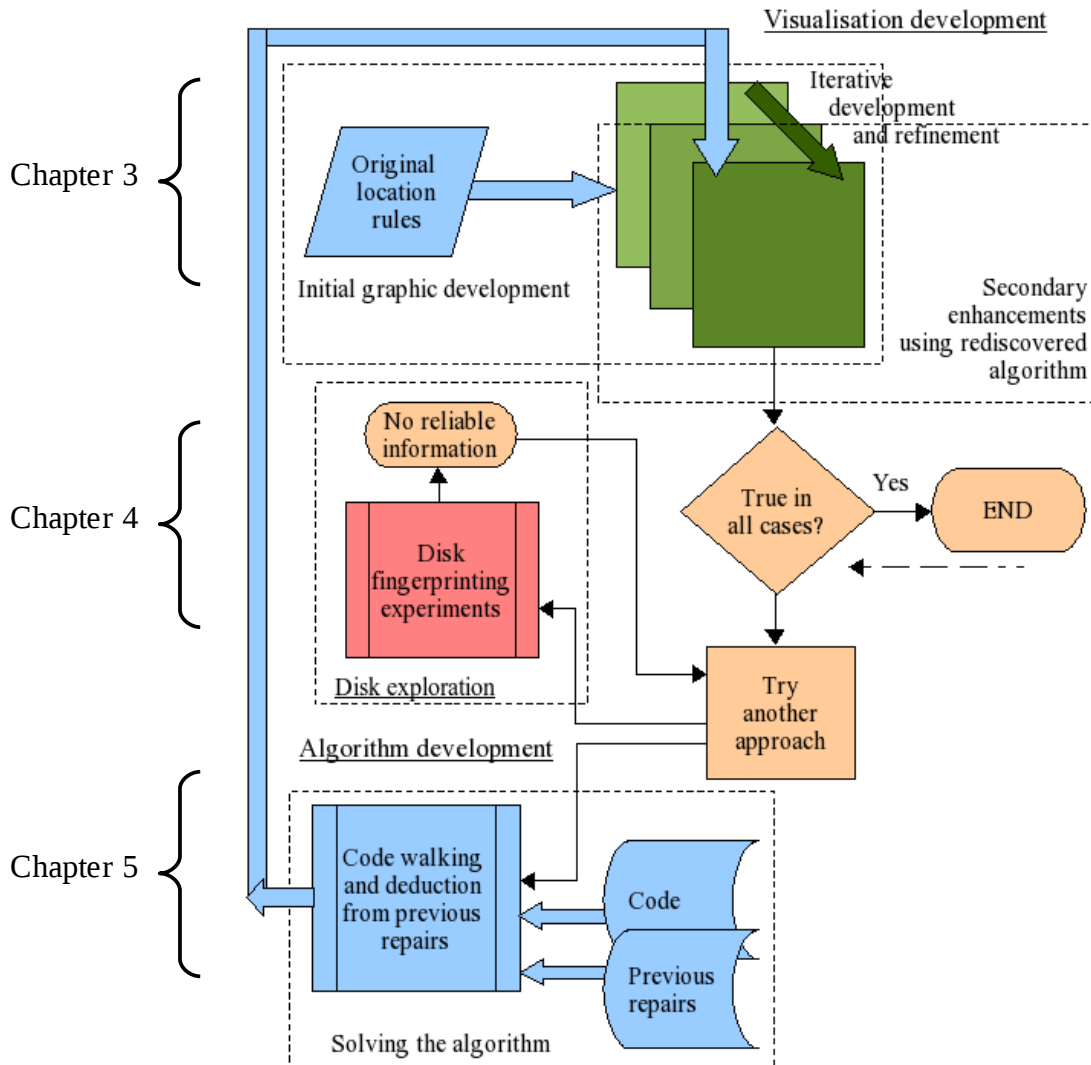


Figure 1.6.1: A diagram of the research processes

In Chapter 4 a different approach was attempted. Here, the data partition was processed sector by sector first to follow data types and then to produce a graphical image of the disk. Although this disk “fingerprinting” gave very clear readable insights into the layout of the metadata and data, it was not able to distinguish between information written to disk during its current operation and data written either in a previous usage of the medium nor an earlier, now irrelevant, system configuration. Early optimism for this process was dampened when a very misleading set of data was processed. Despite the fingerprinting technique not being able to solve this specific problem

it was felt that it was potentially very useful. This is depicted in the middle third of Figure 1.6.1.

In order to work out the correct location and space allocation of the metadata and data the source code for the recorder software application was examined in Chapter 5. This was initially considered to be a trivial task but, whilst a worthwhile exercise, this approach proved more complex than anticipated and eventually working formulas were deduced from an understanding of the nature of the source code methods and a trial and error process applied to historical data. With a reasonable degree of confidence gained from testing that the formula was correct, the formula was added to the at-a-glance processing software described in Chapter 3 which could now detect and present errors in the location metadata to both expert and novice users. This is depicted in the bottom third of Figure 1.6.1 with a line joining into the top of the diagram where the developed at-a-glance image is used and enhanced. So far the formula has proven correct for all instances and the end box is reached. The dotted line beneath the end box allows for the possibility that an instance may be found that does not follow the formula and further refinement may be required. (Although no longer employed at the company contact has been maintained with the senior engineer.)

Chapter 5 saw the realisation of the research objectives and the production of an artefact that incorporated the at-a-glance image with the rediscovered formula. This allowing the disk fingerprinting technique developed in Chapter 4 to be reconsidered in a broader context to see if this method has value beyond the specific problem. These subsequent investigations are presented in chapter 6.

Chapter 7 summarised and considered the work done. Here, enabled and future work was identified as were the contributions made by this research.

2 Background of this Research

2.1 Introduction

This Chapter looks at the background of this research. Section 2.2 considers how a typical Personal Computer (PC) functions and how it stores its data. Section 2.3 looks at the audio recording system and finds similarities between it and the machines considered in the previous section. The following two sections (2.4 and 2.5) consider failure scenarios both for a domestic PC and its file system and for the data partition of the recorder. Common causes of failure are considered in section 2.6.

The data recovery working practices are outlined (2.7) before a description of the disk fingerprinting technique (2.8). The chapter is summarised in section 2.9.

2.2 A Typical PC: Functioning and Data Storage

This section considers a typical PC which consists of a single hard drive that contains the operating system and stored data. Although there may be other devices it is assumed that the PC will boot from this hard drive.

When the PC starts up the attached monitor will display information about the loading operating system. Often the first visible information will be in a textual format giving some information about the hardware components before operating system (OS) specific progress indications are displayed. A modern OS usually has some form of graphical information that indicates the loading progress. By the time the PC is ready to use many tests and processes have completed.

Although reassuring to see information on a monitor, a lot is done before there is any output to the display. The first test that is done is the power supply test. If the voltages are suitable for the

hardware then the next step is to read the information in the Basic Input Output System (BIOS). This information permits the next level of tests to proceed. These checks are often referred to as POST (Power On Self Tests) where the rest of the hardware (other than the power supply) is verified. As these tests may be taking place before the video output is active, any errors detected are indicated as a series of audible beeps sent to the system speaker. These beeps will be hardware specific and usually consist of a series of short or long beeps interspersed with pauses which may be interpreted using the handbook for the machine in question. When no errors are detected a single short beep may be heard.

Once the self tests are completed the PC looks for bootable media, that is media on which bootstrap code is located such as hard drives, CD/DVD drives, USB devices or floppy disk drives. Depending on how the PC is set up it will search for bootable media in each configured boot device. For example, a very common set-up is to attempt to boot from a CD or DVD drive and if no bootable media is present then the hard drive is the next device attempted. This choice of boot sequence allows the PC to boot normally when there is no bootable CD or DVD present, but if there is a problem then a bootable recovery CD/DVD can be placed in the drive and used to repair the problem without the user having to modify settings to change the boot sequence.

Where the hard drive is the bootable medium there is a stub of code located on the first sector (512 bytes) of the drive (sometime referred to as the Master Boot Record (MBR)). This sector contains OS specific code (that will access more complex programs which enable the OS to complete the boot process), and the partition table. The partition table contains information about how the physical drive is divided up (partitioned) into logical drives (partitions). The partition table occupies the last 62 bytes of the first sector before the 4 byte end of data marker (55 aa). The information in the partition table is read by the OS as part of the boot process and if there are obvious errors the PC may not continue to boot.

Although there are exceptions, the most common rules for partition tables are as follows. The partition table has space for four partition entries. These may be primary or extended. A primary partition is a direct reference to a partition while an extended partition is a container that holds information about other partitions. An extended partition contains the information about the partitions it contains while its entry in the partition table (in the MBR) only contains information

about this extended partition.

The partition table contains information about the active status, the type, location and size of each partition. Only one primary partition may be set as being active and it is this flag that will identify the partition as one from which an OS may boot. The type of a partition is its file system type such as FAT16, FAT32, NTFS, ext3 or many others. The location is defined in terms of start and end locations while the size is the number of sectors.

The following is a sample hex dump of a boot sector:

```

00000000  eb 48 90 10 8e d0 bc 00  b0 b8 00 00 8e d8 8e c0  |.H.....|
00000010  fb be 00 7c bf 00 06 b9  00 02 f3 a4 ea 21 06 00  |...|.....!..|
00000020  00 be be 07 38 04 75 0b  83 c6 10 81 fe fe 07 75  |....8.u.....u|
00000030  f3 eb 16 b4 02 b0 01 bb  00 7c b2 80 8a 74 03 02  |.....|...t..|
00000040  80 00 00 80 41 18 01 00  00 08 fa 90 90 f6 c2 80  |....A.....|
00000050  75 02 b2 80 ea 59 7c 00  00 31 c0 8e d8 8e d0 bc  |u....Y|.1.....|
00000060  00 20 fb a0 40 7c 3c ff  74 02 88 c2 52 be 7f 7d  |. .@|<.t...R..}|
00000070  e8 34 01 f6 c2 80 74 54  b4 41 bb aa 55 cd 13 5a  |.4....tT.A..U..Z|
00000080  52 72 49 81 fb 55 aa 75  43 a0 41 7c 84 c0 75 05  |RrI..U.uC.A|.u..|
00000090  83 e1 01 74 37 66 8b 4c  10 be 05 7c c6 44 ff 01  |...t7f.L...|.D..|
000000a0  66 8b 1e 44 7c c7 04 10  00 c7 44 02 01 00 66 89  |f..D|....D...f.|
000000b0  5c 08 c7 44 06 00 70 66  31 c0 89 44 04 66 89 44  |\..D..pf1..D.f.D|
000000c0  0c b4 42 cd 13 72 05 bb  00 70 eb 7d b4 08 cd 13  |..B..r...p.}|...|
000000d0  73 0a f6 c2 80 0f 84 ea  00 e9 8d 00 be 05 7c c6  |s.....|.|
000000e0  44 ff 00 66 31 c0 88 f0  40 66 89 44 04 31 d2 88  |D..f1...@f.D.1..|
000000f0  ca c1 e2 02 88 e8 88 f4  40 89 44 08 31 c0 88 d0  |.....@.D.1...|
00000100  c0 e8 02 66 89 04 66 a1  44 7c 66 31 d2 66 f7 34  |...f..f.D|f1.f.4|
00000110  88 54 0a 66 31 d2 66 f7  74 04 88 54 0b 89 44 0c  |.T.f1.f.t..T..D.|
00000120  3b 44 08 7d 3c 8a 54 0d  c0 e2 06 8a 4c 0a fe c1  |;D.}<.T....L...|
00000130  08 d1 8a 6c 0c 5a 8a 74  0b bb 00 70 8e c3 31 db  |...l.Z.t...p..l.|
00000140  b8 01 02 cd 13 72 2a 8c  c3 8e 06 48 7c 60 1e b9  |.....r*....H|\`..|
00000150  00 01 8e db 31 f6 31 ff  fc f3 a5 1f 61 ff 26 42  |....1.l.....a.&B|
00000160  7c be 85 7d e8 40 00 eb  0e be 8a 7d e8 38 00 eb  ||...}.@.....}.8..|
00000170  06 be 94 7d e8 30 00 be  99 7d e8 2a 00 eb fe 47  |...}.0...}.*...G|
00000180  52 55 42 20 00 47 65 6f  6d 00 48 61 72 64 20 44  |RUB .Geom.Hard D|
00000190  69 73 6b 00 52 65 61 64  00 20 45 72 72 6f 72 00  |isk.Read. Error.|
000001a0  bb 01 00 b4 0e cd 10 ac  3c 00 75 f4 c3 00 00 00  |.....<.u.....|
000001b0  00 00 00 00 00 00 00 00  5f a1 09 00 00 00 80 01  |....._.....|
000001c0  01 00 83 fe 3f 18 3f 00  00 00 9a 20 06 00 00 00  |....??.?.....|
000001d0  01 19 8e fe ff ff d9 20  06 00 e8 69 9b 12 00 00  |.....i.....|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |.....U..|

```

Figure 2.2.1: A sample boot sector

This is the MBR for a Linux PC that uses grub (GRand Unified Bootloader) as a bootloader. The highlighted regions are (yellow) the active primary partition (/boot) and (green) the LVM

(Logical Volume Manager) partition. The LVM manages the main file system and the swap file.

If the partition table is deemed acceptable as the OS continues to load, it may perform a basic check on any partition it is able to recognise. If no errors are detected the boot process continues. Different operating systems have different boot processes. A Microsoft DOS system will seek to find files called "IO.SYS", "MSDOS.SYS" and "COMMAND.COM" and seek to follow the configuration files config.sys and autoexec.bat. A Linux system will run "init" (short for initialisation) and depending on the contents of inittab (its configuration file) will start the applications as defined by the contents of the rc.d. Microsoft's Windows XP system uses NTLDR (NT Loader) which reads the boot.ini file and NTDETECT.COM to read registry hardware configuration before loading NTOSKRNL.EXE (the NT OS kernel). There are many other operating systems; these are just a sample.

The preceding information has been greatly simplified. The exact mechanisms used to boot by different OS is largely irrelevant to the research, what is important is the concept that the data on the hard drive is read and that data is used to control the computer and enable it to function. A far more detailed description of the boot process and partition tables can be found in Sammes and Jenkinson (2007) and specific to the MS system Ray Duncan (1986).

The files used during the boot process must be present, correct and readable. If the system is to read the files it requires then it must have a means of locating the files and this is the primary purpose of the file system. For example, a FAT system is named after its File Allocation Tables of which there are two copies (one backup) at the start of the logical drive. A Linux file system is based on inodes, a system that uses a unique number to identify the information (including name) of each file. A NTFS (New Technologies File System) drive will have MFT (Master File Table), like FAT there is also a backup copy.

Although all these file systems (and others not discussed) are very different they all permit access to stored data. In the normal course of events the PC will successfully boot and the user will use their computer. To create a new text document the user will typically select an icon from their desktop or menu to launch their chosen text editor. This action will initiate the executable program (the editor) or, put another way, this will locate the executable of the hard drive, load it

into memory with any additional configuration information. When the user creates and saves a new document the information will be saved in a file that is written to the hard drive and the file system will be updated to include an entry for this file.

Files are stored on a hard drive in clusters. A cluster is the smallest unit of a drive that can be allocated for a file and it may be (usually) between 1 and 8 sectors in size. The exact size is determined by the size of the partition and the way the drive was formatted. A file is stored in at least one cluster even if it only need a fraction of that space and another file may start in a subsequent cluster. If the first file grows so that it no longer fits in a single cluster then it will be allocated another cluster, but this may mean that the file is fragmented, that is to say that the allocated clusters are not contiguous. The same may occur if the file is very large and there is not a single extent of space into which it may fit. In this case the file must be broken into smaller pieces. The exact mechanisms for this are file system dependent and for the purposes of this discussion it is enough that the possibility of a file being split up is highlighted. Despite this it is common to find that a file can be read directly from the disk by reading sequential sectors.

For the normal operation of a typical PC, the system must have some form of bootable medium (in this case a hard drive). In order for the drive to be bootable there must be a stub of code on the first sector of the drive that will permit larger programs to control the OS specific boot process and a partition table that will indicate where to find this operating system partition. As the PC continues to boot, the file system enables the location of both executable and interpreted data to be used to finish booting and handle subsequent disk storage and retrieval operations essential for normal operation.

2.3 The Recording System: Functioning and Data Storage

The recording system has much in common with the domestic PC. The system uses an active primary FAT partition to boot to its operating system. The operating system partition is a fixed size, irrespective of the size of the recording medium, with the remaining space being used for the second partition that holds the recorded data. These two logical drives may reside on a single

physical hard drive or may be housed on a RAID system. Although RAID technology provides a greater degree of fault tolerance and may provide greater capacity, as far as the system is concerned there are simply two logical partitions (or drives).

Unlike a domestic PC the recorder system does not normally interact with a user and instead it boots to a dedicated software system that controls the specialist hardware required to perform its primary function – namely that of recording audio data. There will be at least one specialist interface card that inputs the telephone signals and data. There is another non-standard interface that allows user commands to be input via a control panel and indications to be output to a small display. Although not always present, most configurations will contain tape drives that are used to create permanent records of the recorded data. There are usually two drives although small systems may use only one and other systems may use a network to backup the data. The whole system may be rack mounted and may not be immediately recognisable as a computer. The recorded data is stored on the hard drive or RAID and periodically archived to tape.

The boot process for the recorder is as described in the previous section. The machine is powered on and the hardware self tests are performed before the hard drive or RAID is located and the stub of code in the first sector is run. The active primary partition is located and the operating system starts to load checking the remainder of the partition table. Although there is a valid second partition, the type identifier is not recognised by the operating system and its presence is ignored. This second partition has no file system and it is written to and read from directly by the recording application. Once the recorder has finished booting this application is run.

When the recorder software starts it reads the first few sectors of the second data storage partition. Here is the information about how the partition is organised and where different regions of data are located. Using the information in this part of the disk, the application is able to locate the different metadata regions. (These regions are frequently referred to as files even though without a file system this term may be considered inaccurate, it represents a concept that is still applicable to that of containing a portion of continuous, related data.)

The incoming audio data is split up into segments of about two seconds duration and recorded in the main body of the data (called disk.dat). If there is only one call in progress the segments of

the call will follow one after the other, but if there is more than one call taking place it is very likely that the next segment will be from a different call. As simultaneous calls are recorded in segments that are interleaved with those of other calls, more information is required if these calls are to be replayed as discrete conversations.

There are two key metadata files that are required for the audio segments to be isolated and replayed. The first is the jump table that records the number of other segments between the segments of a single call. This is simply a sequence of numbers, e.g. 2345211 meaning play 1 segment, jump 2, play 1, jump 3, play 1 etc. This information is referred to as the fat information due to some similarity with the way a File Allocation Table works in a FAT partition.

In order for each call to be recognised there is also an index file, this contains information about different parts of a call. It will identify the start of a call, for example when a customer dials in and selects an option (press 1 for option A, 2 for option B etc.). The next part of a call may be talking to an agent, another part of the call may be put on hold and yet another may be transferred to a different operative. All these parts of a call must be linked together and the index file is a collection of doubly linked lists where each portion has a previous and a next reference. When a call starts the previous value will be null as will the next value when the call terminates.

Although not required for an audio replay, the index file has an associated user defined data file (referred to as the udf file). This file tracks the index file and allows the customer to record additional information about each part of the call, for example which agent handled any portion of the conversation.

There is also a metadata file that permits daylight saving changes to be recorded so that calls can be retrieved in local time.

A typical replay process is as follows: A search is made for a particular call, this may be done by searching for a given date and time. The index information is retrieved and the audio offset into disk.dat is identified. This information is married with the fat information so that once the start segment is found and played, the next segment can be identified and played until all the segments of all the portions of the call are played.

There are two sources of data for a replay. The first is from the hard drive or RAID where the previously described process uses the information still present on the system or it may be performed from tape. Tape drives are used to record long term storage and a heavily used system may have tapes permanently in the drives that are ejected and replaced when full. Such tapes may contain only a few days worth of data. A more lightly used system may have tapes inserted every six months or more. The tapes contain a timespan of audio data (an extract of disk.dat), relevant portions of daylight saving information, fat, index and udf data. When each tape is made, a counter is incremented and the recorded (to tape) data is identified as being backed up.

The data recording partition is laid out as follows (this assumes a typical system with two tape drives).

First is the metadata:

dir.info	this contains the location information about the other files
disk1d.dat	this contains information about daylight saving times
disk1f.dat	this is the fat or jump table information
disk1i.dat	this is the index information
disk1u.dat	this is the customer's user defined information
eventsx.dat	this is a small audit trail
cache.dat	a legacy file no longer actively required
drive1d.dat	daylight saving information for tape drive 1
drive1f.dat	fat information for tape drive 1
drive1i.dat	index information for tape drive 1
drive1u.dat	udf information for tape drive 1
drive2d.dat	daylight saving information for tape drive 2
drive2f.dat	fat information for tape drive 2
drive2i.dat	index information for tape drive 2
drive2u.dat	udf information for tape drive 2
space.wst	a padding file to ensure optimum disk usage

Then the data:

disk.dat	the main body of audio data - this is about 80-90% of the partition.
----------	--

Of these files the information in dir.info should be constant for normal operation. The information it contains defines the start and extent of the other files and is used by the application when writing data to these regions. The drive1X.dat files all contain recorded data. Over time the amount of data stored in each file will increase until the file becomes full (i.e. reached the maximum extent permitted by dir.info). At this point it will go back to the beginning and start to overwrite the oldest data. The rate at which each file is used is different so each will become full at different times. The offset into each file is maintained as a separate variable. These offsets are maintained by the application and stored in its configuration files so that the information is not lost if the recorder is powered down.

The eventsx.dat file contains information about system events such as a reboot, a user logging in, their key strokes and other debug information. The drive files are all temporary data areas used during the creation of tapes. The contents of these areas is not important except during the creation of a tape. The space.wst file is a padding file to ensure that the remainder of the space used by disk.dat is a whole number of records in size.

In summary: During normal recording the recorder will append information to the current position in each of the audio, daylight saving, fat, index and udf files at an offset determined by the application's configuration files from the start of the files as determined by the location metadata, dir.info. When replaying a call the index information is used to find the start of the call as an offset into the main body of the data as determined by the dir.info and the fat information is then used to select the segments to play. The relative offsets to use between the index and fat data are maintained by the application's configuration files.

2.4 When Something Goes Wrong: The Operating System

The previous two sections have looked at how a typical personal computer and the telephone recording system work, this section looks at what can go wrong.

Although the recorder's final operating condition is unlike a typical PC, many of the steps in the

boot and operating process are the same. Both the recorder and a typical PC start out by checking their internal hardware and faults detected at this stage may cause the boot process to halt. If the boot process is halted then it is unlikely that further damage will occur and replacing the faulty component and rebooting is likely to provide a satisfactory outcome. The exception to this is the hard drive itself, although if there is sufficient value in the data on the failed drive then repairs can be undertaken. Such repairs are normally too expensive for a domestic user and a company would need to have a good reason to justify the cost. However physical repairs can be successful even when they necessitate the replacement of motor, printed circuit board (PCB), bearings, preamplifier or any other part of the hard drive to be replaced (Sobey, Orto et al. 2006).

Assuming the hardware is functional, as the PC boots up and the first sector is read, a number of problems may be encountered. The first may be damage to the master boot record code that starts to load the operating system. There are several ways that this could be damaged, historically this region was a popular target for virus writers who would write infectious routines that could reside on floppy disks. If an infectious disk was left in a PC and the PC was set to boot from this drive then it was possible that the master boot sector of the hard drive could be infected. Today such infections are rare as floppy disks are seldom used and anti-virus software is more common. Damage may occur on any portion of a hard drive and the first sector is no exception. Most modern operating systems have the means to repair their boot code, but recovery of the partition table may be more troublesome. In 1998 a virus was written that attempted to overwrite the first two sectors of a hard drive and erase BIOS (Basic Input/Output System) settings. This virus, named Spacefiller, Chernobyl or CIH, triggered on April 26th, 1999 and the 26th of every month thereafter. Initially there were many drives with destroyed partition tables and a few months later there were many utilities released that were able to recreate partition tables.

Assuming that the initial boot code is successful one of the next possible areas for trouble is the file system. If there is damage to the file system the operating system may not be able to find the files it needs to complete the boot process. In such a scenario it is possible that a recovery CD may be able to repair the file system structures and enable the computer to function once more. It is also possible for file system damage to be sufficiently minor that only a few files are lost and the user may not even notice that they are gone.

Although technically not file system damage, another reason for files being lost is that a user deletes a file they later wish to access. When a file is deleted the pointer to the file is usually left intact, however the area on the disk pointed to is marked as unused. Data loss through this type of user mistake is sufficiently common. In fact many free utilities are available to recover files after they have been deleted. D (2007) provides a guide to such a recovery process.

Physical jarring, unexpected loss of power, viruses or a deteriorating hard drive are among the reasons why data may be lost. Whatever the cause, the extent of data loss or corruption will vary. Sometimes only part of a file may be damaged. For example, if portions of a Microsoft Word document are corrupted then it may not be possible to view the document with Microsoft Word; however opening the document with a plain text editor may allow for some or all of the textual contents to be extracted. Although all formatting and any inserted images or diagrams would be lost with this technique, being able to open the textual contents of the file normally represents an improvement over recreating the document from scratch (but obviously not as good as having a reliable backup).

Most files are stored to disk as sequential sectors of data. Even where file system damage has occurred a simple search of the drive, sector by sector is likely to find the information eventually, providing it has not been overwritten by newer data. This process is time consuming and not all operating systems will permit direct access to a drive, but it is often possible to use a “live” Linux CD (one that boots into a usable version of the operating system) and search the media using this temporary system. Once found the file can be copied to another storage location such as a flash memory stick or another drive if present. (Recovered data should never be saved to the drive from which it was recovered, as it is possible that the saved data may be written over the data that is being rescued.) Even where files are not contiguous and a file was stored in several chunks, it should be possible to find these chunks and piece them back together. Although this form of data recovery may be beyond the expertise of average users, it is not a difficult process and there are occasional articles on such techniques in computer magazines (for example, see Farmer (2001)).

Although all the problems considered above could effect the recorder it is unlikely that any of the recovery techniques discussed would be used. The primary partition contains only the system software and for the most part this is not considered valuable as the data on this partition can be

reinstalled. The exceptions are the state and configuration files. The configuration files contain information about how the recording system has been set up, e.g. the number and type of tape drives installed for creating long term backups or the number and type of cards used to interface to the telephone system. The state files capture the state of the machine with information such as pointer offsets or the number of the last backup tape created. Although it is helpful to access this information, these files exist for use by the application software rather than forming part of the stored data and they can be recreated by careful examination of the stored data.

Usually it is not worth the time searching the medium for the configuration files as they can be recreated. If the partition table were overwritten it would be fairly trivial to edit back the partition entries by copying the data from another machine using a disk editor. If this were the only problem then this repair would permit access to the useful files. Another quick repair possibility if the file system is damaged is to access the partition from a Linux PC and run the `dosfsck` utility. This tool takes only minutes to run and may save time later.

Often a recorder will use a RAID system to provide some fault tolerance. In such a system a single hard drive may fail with little or no impact. Sometimes the impact is so minimal that the failure goes unnoticed until a second drive fails. In this case the drive must be physically repaired before the RAID can rebuild itself. If a rebuild is successful there may be no repair required.

If there are problems with the application partition then one of the quickest ways to repair the operating system is to `dd` the contents from another recorder into an image and write this image over the damaged partition. Although the primary partition closely resembles a typical PC configuration a major difference is that minimal data is stored here and although it is useful to access this drive it is of little significance if it is lost.

2.5 When something Goes Wrong: the Data Partition

Under normal operating conditions the location metadata identifies the start of the files and the recorder's application keeps a record of how far into each of the files is the current recording position. New data is recorded after these positions and existing data can be replayed from behind

these positions.

If this offset information is lost, for example if the application partition is damaged, then it is possible to recreate the information. If the recorder has not been in use for a long time then the metadata regions will not be full and the end of the written data will be the offset. The situation is more complex when the recorder has been in use for a longer period of time and some or all files have become full and are overwriting the oldest data as they re-traverse their allocated space. Although more difficult, the current offsets can still be deduced. Embedded into the main body of audio data are occasional metadata blocks from which time and date may be decoded. This information can be used to determine the offset of the latest audio information recorded (in `disk.dat`). The index files also contain the date and time of the call and this too can be used to find the latest recording of this data. As the user defined data has the same offset as its companion index, if the index offset is known then so is the udf offset. The fat (jump table) files are more difficult. Here the chain of jumps is traversed to find the durations of the calls. This length of call table is then compared to the lengths of calls in the index data and when a match is found the offset into the fat data can be deduced.

The most common repair scenario is where the recorder either resets itself (for reasons unknown or unreported) or is manually reset. Here the offsets are zeroed and the recording goes back to the start of each of the files simultaneously.

In a repair the initial assessment is an analysis of the various metadata files. To do this the information in `dir.info` is read and decoded. By using the decoded location information the key file regions are copied into (real) files for analysis. The initial analysis examines the files that contained date/time information. Figure 2.5.1, of a typical history, shows the type of information that may be discovered. The numbers in this file indicate incrementing date and time values. These are greatly simplified for the purpose of this explanation as in reality there may be many thousands of entries.

This figure shows that the oldest data (right) that has filled the allocated region (records 12 to 16 still visible). These oldest records were in the process of being overwritten with newer data, records 17 to 24, although only record 20 onward is still available as a reset occurred after record

24 was written and the recording jumped back to the start. This reset went unnoticed and records 25 to 27 were recorded.

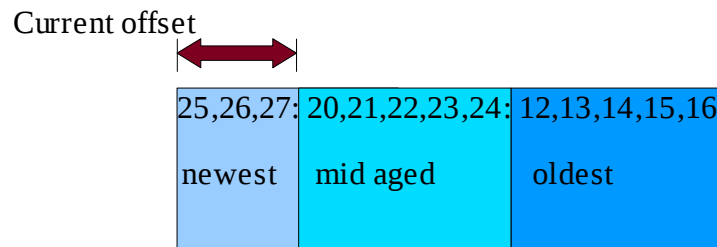


Figure 2.5.1: A typical history

This type of data dislocation is the most common type of repair. It is probable that backup tapes will have been made of all the data up to (for example) record 23. This leaves 24 and 25 to 27 to be backed up. (Although it is always possible that data, which has not been backed up, may be overwritten in practice this is unusual and only occurs if there have been several resets since the last backup.) In a typical repair the customer will want their data archived to a backup tape. The standard way of achieving this is to copy the newest data to the position it would have been recorded to, had there not been a reset, and then recalculate the offset. See Figure 2.5.2 below.

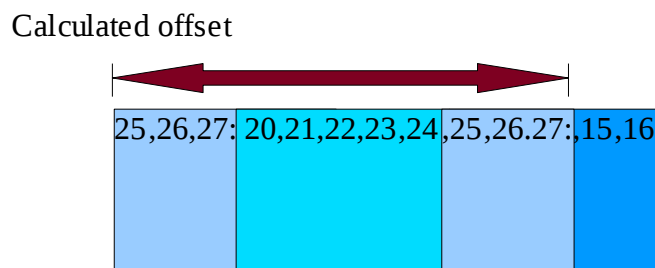


Figure 2.5.2: A repaired file

Once all the files have been similarly repaired and all the correct offsets calculated, tapes may be inserted and the required backup made. (This explanation ignores the fact that the underlying reason for the reset may have produced other errors and repairs may also be required on the relocated data.)

Although it is possible to manipulate different aspects of the recording the only time the location metadata should be changed is if a recorder's owner wishes to make changes to the amount of information stored in their user defined data. As far as it is known, all other reasons for changes

are errors.

For such a change the following steps occur:

1. The front line engineer will define the new information fields.
2. The changes will be stored and the recorder will be rebooted.
3. As the recorder recognises that fundamental changes have been made, it will recalculate the recording media layout and rewrite the location metadata (dir.info).
4. All offsets will be set to zero, meaning that all files will now start recording at the beginning of their newly allocated positions.

Changing the user defined information is not a common occurrence, most customers would decide what information was required and this would remain constant over the life cycle of the recorder. The procedure is not usually troublesome as prior to the changes backups should be made so that there is no data lost. In this scenario the changes are made through a deliberate set of actions and the application is expected to behave in a predictable fashion. If backup tapes are not made before the changes then, if the recorder is returned to the data recovery laboratory, it should be possible to return the user defined information to its previous size and recalculate the offsets. Tapes could then be made before the new changes were reapplied. As the reason for the reapportioning of the medium was known the application could be trusted to recreate the same layout for the same inputs.

Although very rare, a similar situation can occur for less obvious reasons. No software is perfect and, although bugs are fixed when discovered, the recorder application has historically (and passably currently) encountered situations in which a parameter is modified and the machine rewrites the location metadata and resets the recording offsets. If the recorder is located in an equipment room and no one observes the machine rebooting, it is possible that the event may go unnoticed until the backup tapes are examined. This is possibly the worst case scenario, as the formula by which the recording medium is apportioned is unknown, the changes in the location metadata may not be detected. The recorder will not attempt to correct such a change on subsequent restarts.

Figure 2.5.3, a misaligned file, shows a typical metadata file that has filled its entire region and is in the process of overwriting the oldest data. In this case no data has been recorded after the recorder failed. The blue file shows records incrementing and at the value of 40 the maximum extent is reached and the oldest data is overwritten at the start of the region where the data increments to reach record 49. This could be a very simple repair, however if the cause of the failure had been an event that modified the dir.info data then the wrong region of data may be captured as indicated by the peach region.

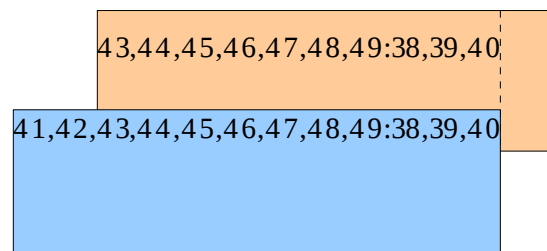


Figure 2.5.3: A misaligned file

The incorrect capture would indicate that the record 40 was reached. As the first record at the start is 43 then a logical explanation would be that the recorder wrote 41 and 42 before a reset occurred and the pointers went back to the beginning with record 43 and 44 onward overwriting the missing information. The assumption of a reset may seem a non-intuitive conclusion, however with several files showing similar symptoms and reset being a common reason for a repair, it is not an unreasonable assumption.

In theory wrong offsets could be calculated to compensate for the wrong start location, but these would mean that the relative offsets between files were incorrect. It is believed that a correct repair cannot be made until the location information is corrected. Prior to this research it was not possible to determine if this information was correct and unless there were obvious errors the repair would proceed on the assumption that it was correct. Fortunately this was usually the case.

A recorder being reset is usually a symptom of a problem. Very often there is a reason for the reset and this may need correcting before the repair can be effected. Sometimes the data or metadata is damaged or changed in a way that is not understood. For example, a repair was encountered where there appeared to be 20 Gigabytes of audio data missing. The reason the data

appeared to be missing was that the validation of the data was done by searching for embedded metadata segments. As these were not found in this region of the disk, it was thought that the audio data was missing. It transpired that the check for this embedded metadata was performed by checking only the sectors that were expected to hold this information. A closer inspection showed that the data on this region of the disk was a sector behind its anticipated location and as a result the metadata structures were not in the sample regions. The mechanism by which the data was written to a sector after its anticipated location is not known nor is it understood why, after writing 20 Gigabytes, this should correct itself.

That the recording partition has no file system means it avoids a layer of complexity, however this choice means that the data and metadata are written to the disk as the application sees fit. If there is any form of problem in the recording system there is a possibility that data may be written to any portion of the recording medium. Fortunately the recorded information is usually in a predictable location or, if not, it can be located. Only one instance, during the employment as a data recovery engineer, was metadata found to be missing as discussed in Chapter 5.7.

Once the data structures are believed to be correct, the repair must be tested. For this the recording medium will be connected to a recorder with either compatible hardware to the original or with configuration changes made so that the machine becomes compatible. The machine is powered up and then, using the front panel controls, a search is performed and a call selected for replay. When the replay option is selected a replay progress bar will appear on the display and the call will be replayed. If everything is successfully realigned then the call should sound correct and will finish when the progress bar reaches the end. If one call plays successfully then further calls are selected. Sometimes the nature of the initial fault condition must be taken into account, for example if the recorder was subject to frequent spontaneous reboots then it will be unrealistic to expect perfect audio output as the system cannot record while it is down or rebooting. In this situation gaps in the audio are to be expected.

If the data partition has been successfully repaired, creating backup tapes is simply a matter of inserting blank tape(s) and waiting while the data is written to these tapes.

2.6 Causes of Failure

The primary types of causes of loss of data have been identified as the following:

1. Hardware faults.
2. Software faults.
3. Unfortunate front line support.

The first, hardware faults, covers such events as physical failure of the recording media. This may be the failure of a hard drive's motor or bearings necessitating a specialist clean room repair. Problematic power supplies may cause frequent reboots resulting in the loss of recording while the machine recovers. Memory errors may cause unpredictable data computation and network cards "hanging" in some configurations may suspend the operation of the machine.

Such hardware problems are common to almost any computer-based system, but one recorder specific, recurring problem identified was an ill considered routing of SCSI ribbon cables adjacent to the power supply feeds. (SCSI cables can be subject to interference that may impact on the SCSI devices attached.) It is believed that this was the source of sporadic corruption found on some (SCSI) recording medium. An example of a recorder with bursts of corruption is described in Chapter 4. Investigations into SCSI I/O operations are still ongoing, but where this problem occurs the bursts of corruption seem to be randomly distributed. The problems that arise from this may be as minimal as a 2 second burst of noise in an audio recording; as catastrophic as losing the location metadata or somewhere in between depending on the damaged file. For example, damage in either index or fat files will render the associated calls unable to play correctly.

No application of any size can ever be completely free of software faults. When an observed problem is recognised as having a software cause, time and effort is expended in rectifying the error. Sometimes fixing one problem can cause unexpected side effects and the process of identifying and rectifying the resultant features begins again. Some software problems require specific circumstances before symptoms arise. One such problem was an overflow caused when the number of channels being recorded multiplied by the number of years in operational use

exceeded a buffer size.

Front line support is an area that is not completely understood. It is possible to fix a problem and release an upgrade only to later discover that rumours and superstition have meant that the fix to a known problem is not applied. Despite extensive testing the channel/year fix (for the problem described above), it was rumoured to cause a system reset even though no supporting evidence was offered. Repairs still came to the laboratory when this problem occurred and contrary to instructions the fix was not applied.

It is not uncommon that one of the last events to be captured (on the recorder's audit trail) is of someone logging into the system with the administrative password. When logging in with this level of privileges it is possible to change all of the configuration options, from user defined fields to the identity of the machine itself. Despite such evidence the fault report may state that the recorder “just failed”. While some sympathy may be felt for the front-line maintainer it is frustrating to realise that potentially useful information is being withheld and that the repair report can only state the time and date of the recorded login relative to the failure.

The very essence of this data recovery work is that each job is new unexplored territory and it is likely to have at least one aspect that has not been seen before. A secondary task for most repairs is to perform an RCA (Root Cause Analysis). Unfortunately on most occasions not even an educated guess can be offered. On the occasions that a diagnosis can be made it is usually an improbable sequence of actions performed by the front line engineer. These actions are highly unlikely to be replicated. It is this unpredictability that provides the need for this research. If there were a set of predictable faults or types of fault then these could be identified and the repair could proceed accordingly. With no assumptions possible before a repair starts every step must be confirmed. Although the location metadata is usually correct it cannot be assumed that this is the case and without verification all other actions risk being wrong.

Although the criteria for determining correctness is a qualitative process it combines the astonishing ability of the human ear synchronised with a corresponding visual input to produce a quantitative analysis of either everything is correct or the repair is not yet complete.

2.7 Company Work Practices and Data Recovery Future Plans

While working as a recovery engineer two basic disciplines were observed. The first was that all data recovery attempts began with the cloning of the medium and the recovery attempts were performed using the copy. This ensured that any mistake made during a recovery attempt did not destroy the original data. With the original data untouched it was always possible to take another clone and reattempt a repair.

The second discipline was that each recovery engineer kept a log of all the steps taken. This log was used if a mistake was made so that the successful steps leading up to the mistake could be reapplied (on a fresh clone). It was also useful if another engineer had to take over a repair and could provide the basis for a discussion if things do not proceed as expected. Although highly unusual it is possible that a conversation from a recovered system could be deemed as evidence in future legal proceedings. As this may be many months, possibly years after the repair, the log would provide the engineer with the confidence to say that their repair contained the same data as was originally recorded.

Working in the laboratory was a specialist area and the work was performed in isolation from other company sections. This did not mean that engineers were left unsupervised, but it was recognised that the nature of the work meant little could be put into formal procedural documentation. This did not prevent the company from having long term plans for a Recovery Studio. This was to be an application that would provide a basis for automating the data recovery process.

Ideally every front line engineer would have the skills to recover lost data at the time the problem was identified, but data loss is too rare an occurrence for such a training exercise to be cost or time effective. Instead a data Recovery Studio is envisaged that will allow a front line engineer to progress through a recovery process either solo or with remote assistance. Work on this recovery studio has not yet begun. Since almost every data recovery task is different and there is a lot of analysis to be done before any development can begin.

Over the years data recovery engineers have developed many small dedicated purpose utilities to help examine, diagnose and repair recording media sent for recovery. For example, there is a utility for examining the dir.info region and using the information there to capture the index data to a file in a working directory and another for converting the binary information to a human readable textual format. If errors are found in the text file these can be corrected before another utility is used to convert this back into a binary form. Once a corrected binary is produced all or part of it may be copied back to the medium using the information extracted from dir.info and the Unix dd command. (This last part is normally done by hand rather than an automated process.)

The purpose of the recovery studio is to provide a framework for these utilities and others, some not yet written, to be combined into a single relatively automated GUI (Graphical User Interface) based system. The requirements for the Recovery Studio describe this concept and identify the preferred software development languages. These consist of the Trolltech QT4 graphic libraries for developing the GUI, shell scripts, python, C, C++ and sed/awk scripts for performing the underlying tasks.

A large portion of the requirements specification is a list of technical problems that need to be solved before the work can begin. These tasks are mostly in the form of a list of small program specifications many of which have already been written (or solved). An example of a doubly linked list and the task of working out the base starting point is included in appendix B to give an indication of the nature of these puzzles.

Another identified requirement is for some form of graphical representation of the recorded metadata and data so that a drag and drop, initiated from the GUI, could be translated to moving files to, from or around the recording medium. It is recognised that a graphical representation, if it is to represent all the data could not provide the fine detail needed to select a single record or group of records. However such a representation could be used to launch a slider (or similar) that could permit a more exact selection to be made. More pertinent, to this research, is that the same graphic could be used to highlight suspected inconsistencies or errors that could alert an engineer to the presence of a problem.

The requirements include the need for verifying or recreating the location of the metadata and the

following comments are included in the requirements documentation: “Note: this requires a detailed synthesis of the “XXX” formula which is non-trivial.” (Where XXX is the application name.) The search for this formula has been previously attempted without success. Solving this formula will mean that the location metadata can not only be read and decoded, but also verified before the metadata regions are captured into files for further processing. If there is a high degree of confidence that the metadata has been captured correctly then there is scope for automating its evaluation and the point at which a human needs to intervene moves further into the recovery process.

2.8 Disk fingerprinting

A large number of the data recovery process use the Unix `dd` command. This utility copies data from a source to a destination. The source may be a device such as a hard drive or it may be a file. When cloning a RAID or hard drive prior to recovery work a variant of `dd` was used. If no output is given then `stdout` (the command prompt) is used. This means that the output from this command may be piped to other Unix commands. Very often there may be a need to check and see if the data being copied is the correct data. To an experienced eye piping the output to `hexdump` can provide the means of checking. For example `dd if=/dev/sda count=1 | hexdump -Cv` will copy all the data on the first sector of the first drive to `hexdump` formatted for ASCII as seen previously in Figure 2.2.1.

Although the ASCII representation is probably the most common use of `hexdump` it has other switches. If the `-d` switch is used then the output is in decimal, `dd if=/dev/sda count=1 | hexdump -dv` produces the following:

```
0000000  18667  04240  53390  00188  47280  00000  55438  49294
0000010  48891  31744  00191  47366  00512  42227  08682  00006
0000020  48640  01982  01080  02933  50819  33040  65278  29959
0000030  60403  46102  45058  47873  31744  32946  29834  00515
0000040  00128  32768  06209  00001  02048  37114  63120  32962
0000050  00629  32946  23018  00124  12544  36544  36568  48336
0000060  08192  41211  31808  65340  00628  49800  48722  32127
0000070  13544  62977  32962  21620  16820  43707  52565  23059
```

0000080	29266	33097	22011	30122	41027	31809	49284	01397
0000090	57731	29697	26167	19595	48656	31749	17606	00511
00000a0	35686	17438	51068	04100	50944	00580	00001	35174
00000b0	02140	17607	00006	26224	49201	17545	26116	17545
00000c0	46092	52546	29203	47877	28672	32235	02228	05069
00000d0	02675	49910	03968	60036	59648	00141	01470	50812
00000e0	65348	26112	49201	61576	26176	17545	12548	35026
00000f0	49610	00738	59528	62600	35136	02116	49201	53384
0000100	59584	26114	01161	41318	31812	12646	26322	13559
0000110	21640	26122	53809	63334	01140	21640	35083	03140
0000120	17467	32008	35388	03412	58048	35334	02636	49662
0000130	53512	27786	23052	29834	47883	28672	50062	56113
0000140	00440	52482	29203	35882	36547	18438	24700	47390
0000150	00256	56206	63025	65329	62460	08101	65377	16934
0000160	48764	32133	16616	60160	48654	32138	14568	60160
0000170	48646	32148	12520	48640	32153	10984	60160	18430
0000180	21842	08258	18176	28517	00109	24904	25714	17440
0000190	29545	00107	25938	25697	08192	29253	28530	00114
00001a0	00443	46080	52494	44048	00060	62581	00195	00000
00001b0	00000	00000	00000	00000	41311	00009	00000	00384
00001c0	00001	65155	06207	00063	00000	08346	00006	00000
00001d0	06401	65166	65535	08409	00006	27112	04763	00000
00001e0	00000	00000	00000	00000	00000	00000	00000	00000
00001f0	00000	00000	00000	00000	00000	00000	00000	43605

This is the same information as before except the information is now in decimal and there is no textual presentation. The column on the left is the offset in hex into the sector. The decimal values can range from 0 to 65535. The sum of all the values for this sector is 6735643.

The developed disk fingerprinting technique is very simple. It takes the sum of a sector as calculated above and records its offset. For example, the above is offset of the sector is 0 and sum is 6735643. The same calculation is performed on all the sectors of interest and a graph is created using gnuplot (gnuplot 2009). Gnuplot is a powerful command line driven scientific graph generator. The reason this graph generator was chosen was because of its availability and ease of use. There is no reason why another utility should not be used to generate the same graphs.

Gnuplot allows the user to set the output to an image. For all of the images used in this thesis the output was set to png (Portable Network Graphics) so that images could be generated that could be inserted into the document. If this option is not selected gnuplot will by default open a window for the graph. This window can be resized, zoomed in and otherwise manipulated to allow examination of the graph. In the following screen shot of gnuplot, Figure 2.8.1, the command line from which the utility was launched can be seen in the background as can the single command `plot ratcatcher-fp.txt`. The resultant pop-up window and the generated graph are

shown in the foreground. The data used was partial fingerprint data from a Linux laptop called RatCatcher.

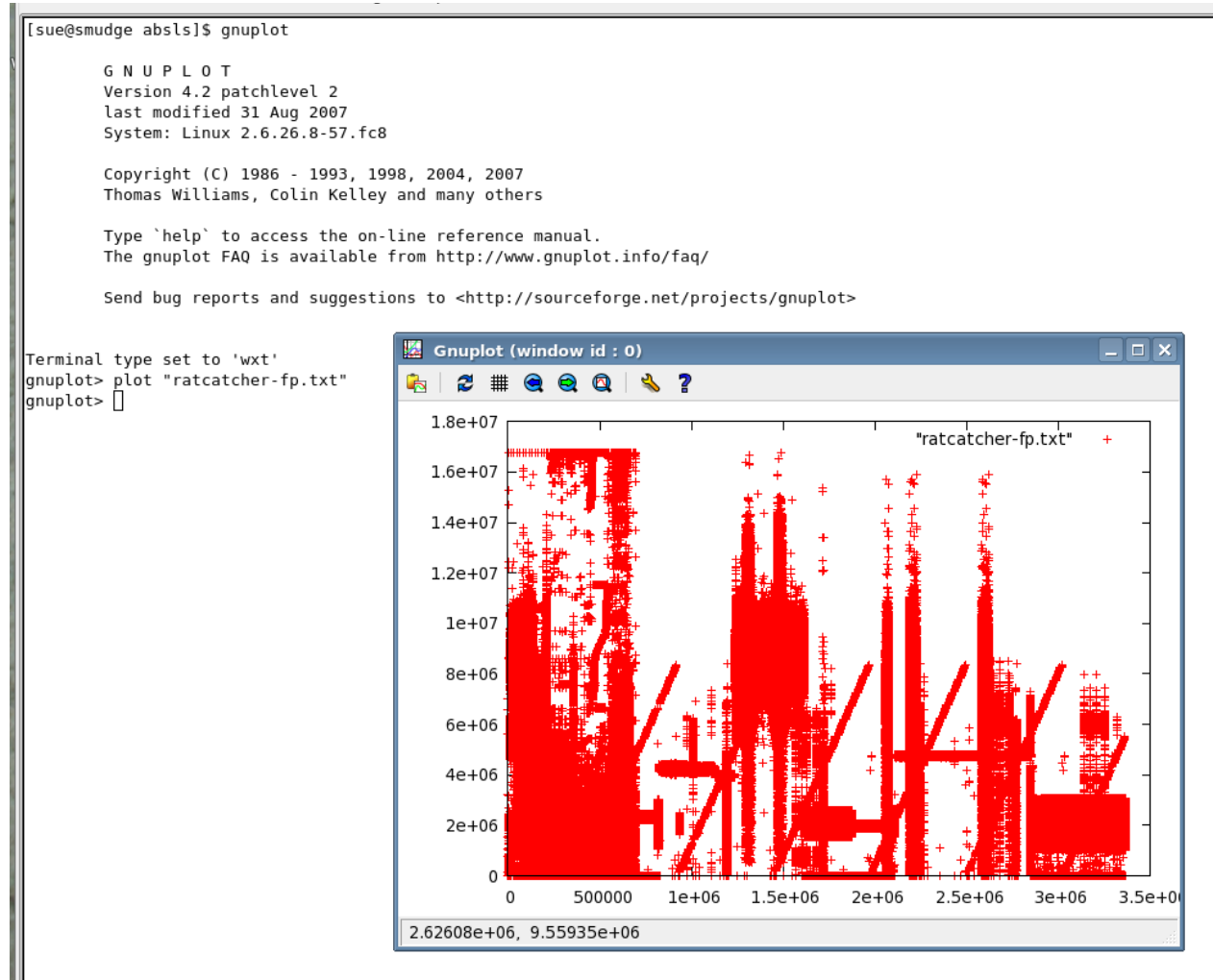


Figure 2.8.1: gnuplot basic

The above figure shows how very simple it is to display about 50MB of data. In order to make the images used in the document the following commands were used:

```
gnuplot> set term png size 800, 270
Terminal type set to 'png'
Options are 'nocrop medium size 800,270 '
gnuplot> set output "ratcher.png"
gnuplot> plot "ratcatcher-fp.txt"with dots
```

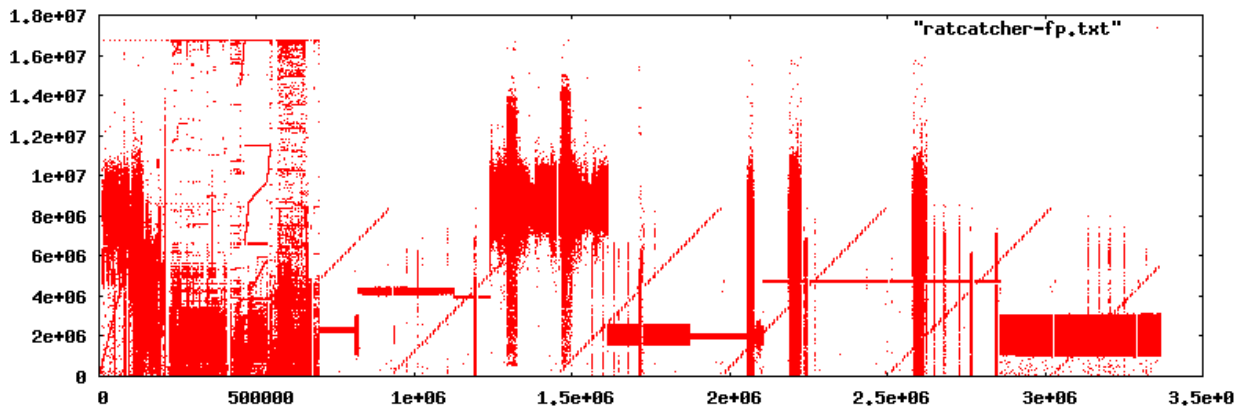


Figure 2.8.2: Linux laptop fingerprint

The first line sets the output to type png with a size of 800 x 270 pixels. In the following two lines are the confirmation that the command has been accepted and available options. The next step is to provide the name of the output file and the final line modifies the default plot of a cross to a dot. Figure 2.8.2 shows the refined image of a Linux laptop. This fingerprint shows the boot partition and the start of a LVM volume. The structures seen in this image are typical of the fingerprints taken from other Linux machines. Chapter 6 explores this in greater detail.

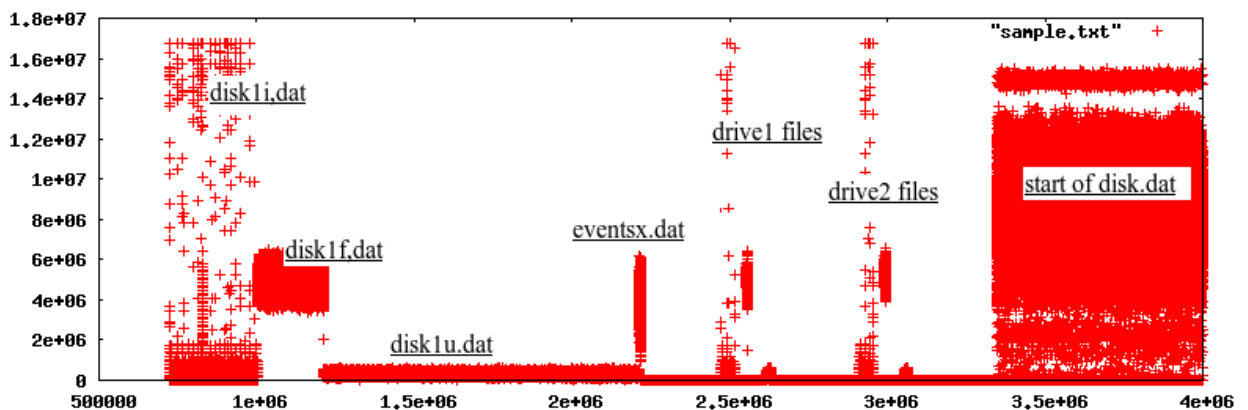


Figure 2.8.3: Start of a typical data partition

Although the fingerprinting technique has considerable potential for domestic PCs and data recovery in general, it has proved to be extremely useful as a diagnostic aid in this specific data recovery situation. Figure 2.8.3 shows the start of a typical data partition. The basic fingerprint

produced from iterating the metadata and the start of the data from a normal working recorder. Chapter 4 contains far greater analysis of this type of information. The identifiable regions are labelled.

Given that the basic concept of plotting the sum of a sector is so very simple, that different types of metadata and data should produce such visually different patterns is considered most serendipitous.

2.9 Chapter Summary

This chapter has looked at the basic functioning of a typical personal computer and how it functions when all is working as expected. Many of these functions are the same for the telephone recording system as its operating system is very similar to a domestic PC.

Having looked at how a functioning PC works consideration is then given to the machine that does not work. As the types of problems that can effect a PC are the same for the recorder system the two are considered as being effected by common problems. There is a big difference in the recovery techniques as a domestic machine is likely to contain valuable (to its owner) data while the recorder only contains a few useful but not essential files. To this effect, while time and energy may be invested in recovering data from a PC, it is unlikely that more than a brief attempt would be made to restore the application partition of a recorder.

The data partition of the recorder contains valuable information. Its structures and its failure modes have been examined and the reason why correct location metadata is so important to a repair has been described. It would have been useful to describe typical repairs, but the fact that the system is so reliable means that when a failure does occur it is usually unique. Simply put there are no typical faults. Despite this lack of typical faults a recorder reset is common enough to warrant a description. The steps needed to recover from a reset give some idea into the working of the recording equipment.

This chapter has also considered the working practices of the company and for data recovery in

general. One of the company specific considerations was the Recovery Studio project. The requirements for this project have influenced the direction of this research to a large degree but given that the company permitted almost unlimited access to their data and equipment this was not considered unreasonable.

During the research process a disk fingerprinting technique was developed. This has also been described here although greater detail of its use will be included in the second half of this thesis where the research events are described.

3 Visualising the data structures

3.1 Background

The events described in this chapter took place in the early days of the employment. At that time no documentation existed for most of the recovery procedures and one of the employment duties was to write the documentation. Usually a PhD student will read the available literature and then work forward from the understanding gained from such writing. In this case the initial understanding was gained orally from the senior engineer or had to be deduced from first principles. Once an understanding had been gained the documentation was written.

Although there were usage instructions and man pages for most of the utilities that were used, the only formal documentation that existed, in a form management would recognise, was that of the requirements for the Recovery Studio. It was this document that was used as a template from the subsequent documentation of repair procedures. The requirements for the Recovery Studio represented a collection of problems and puzzles that provided useful insights into research areas that would be challenging and useful to the company.

At this time the understanding of the repair process was incomplete. Only a couple of repairs had been performed before a problem was encountered where the location metadata was incorrect. The initial assessment process was a basic shell script that took information from the location metadata and copied regions of data into files before decoding anything with a date/time or serial number information. During this repair the anomaly was detected by the assessment script, but the error was output to the console and had scrolled off the screen without being read.

The error was spotted by the senior engineer while supervising the repair. The location metadata was corrected and the recovery job continued. Afterwards an examination of the initial script was made and the process by which the error was detected was isolated. The detection was made by a finite set of rules and it was believed, at the time, that all location errors could be detected. The initial problem seemed to be not the detection but the presentation of detected errors.

Having read the Recovery Studio requirements it seemed that the need for a visualisation of the data and metadata could be combined with a means of detecting and presenting the location metadata. Although the assumption that all errors could be detected in this way was somewhat naive this chapter documents the development of the at-a-glance representation that could detect a subset of possible errors.

3.2 Introduction

This chapter describes the first attempt to solve the problem of verifying the location metadata. As the original understanding was incomplete and it was believed that there was only a set of detectable errors, the focus is on presenting the metadata and data information to engineers in an at-a-glance image that would alert them to any detected errors.

Section 3.3 describes the starting information in the form of an `abslsr.txt` file including a sample file. The next section (3.4) looks at the reasons for selecting the software language and graphic format chosen before the initial first attempt is made (3.5) where the limitations of a linear representation become apparent. Logarithmic scaling is then attempted (3.6) and this approach permits greater insight into the data allowing more pertinent information to be revealed in section 3.7. With a good representation of the files, extra intelligence is added so that errors can be presented to the observer in section 3.8 before the achievements of this chapter are considered in the final section (3.9).

3.3 The `abslsr.txt`

The `abslsr.txt` file is an output from the initial processing of the `dir.info` region of the recording medium. At the time this work was undertaken it was believed that the `abslsr.txt` information was always fundamentally correct and that there existed a finite set of errors that could be detected and correct values deduced. Although this assumption was subsequently proved to be wrong there

was also a need to represent information to a novice engineer in such a way that errors in the location metadata were immediately visible and in this respect the progress made here is of long-term use. It is feasible that the resultant image may also be used as a basis for dragging and dropping files to/from the recording medium as part of the future Recovery Studio development. As this falls outside the scope of this research then care can only be taken to preclude as few options as possible.

This stage sought to generate an at-a-glance image that could convey all the relevant information to an engineer in a single graphic. Then they may be reassured that all is correct or be alerted to a problem that may either be overridden by local knowledge or a proffered solution could be accepted. Even though the underlying error detection code is unable to detect all errors (the ultimate objective), it is sufficient at this stage to note the creation of an at-a-glance image that is able to represent the errors known at the time of its creation. The image is revisited in chapter 5 after the underlying formula for the metadata layout has been rediscovered and much greater fault information is available.

name	bytes	start_address	sectors	records
dir.info	4096	722925	8	4096
disk1d.dat	2000	722933	4	2000
disk1f.dat	859523672	722937	1678758	214880918
disk1i.dat	644642304	2401695	1259067	13430048
disk1u.dat	4123024736	3660762	8052783	13430048
eventsx.dat	1245184	11713545	2432	65536
cache.dat	130023424	11715977	253952	130023424
drive1d.dat	2000	11969929	4	2000
drive1f.dat	41943040	11969933	81920	10485760
drive1i.dat	31457280	12051853	61440	655360
drive1u.dat	201195520	12113293	392960	655360
drive2d.dat	2000	12506253	4	2000
drive2f.dat	41943040	12506257	81920	10485760
drive2i.dat	31457280	12588177	61440	655360
drive2u.dat	201195520	12649617	392960	655360
space.wst	142336	13042577	278	142336
disk.dat	433768017920	13042855	847203160	211800790

Figure 3.3.1: A typical abslsr.txt

The initial automated scan will produce an abslsr.txt, a text file listing the file names, their size in bytes, start address, number of sectors (rounded up if only part used) and the number of records. A typical abslsr.txt is shown in Figure 3.3.1. The objective is to present this information to the

recovery engineer in a more intelligent visual format.

Of these files the first, `dir.info`, is the location metadata with the subsequent four files containing the metadata for the main audio recording. The `eventsx.dat` is an audit trail of the most recent events. DriveX files refer to the tape drive X and in Figure 3.3.1 it can be seen that there are two tape drives. This represents the most common configuration but there may be up to eight drives installed or there may be none. `Disk.dat` is the main body of the data with `space.wst` and `cache.dat` being padding files to ensure the optimal use of the recording media.

3.4 Design Decisions

As there is a great deal of variation in the file sizes some consideration was given as to how to represent this information. If the largest file were to be displayed in a reasonable scale then the smaller files would be barely visible. If the smaller files were appropriately displayed then the largest file would require considerable scrolling before its trailing edge could be discerned.

To enable a viewer to see both the macro and micro information, a form of zooming in and out was considered necessary. If a zooming mechanism was to be used then any image must either be redrawn for each step of the zoom range or an image format that natively supported such scaling must be used. Of the two options a scalable image format seemed to offer a simpler way forward and this option was chosen. It should be noted that the option of generating multiple views of the same information was not completely discarded as practical implementation considerations could still make this a more efficient option.

The decision to use a scalable image format meant that the choice of image type was limited. This option predisposed that the image was a form of vector graphics (as opposed to raster or bitmap image). SVG (Scalable Vector Graphics) and PostScript were considered. SVG was chosen as it would render in both the Opera and Firefox Internet browsers; it was a relatively simple XML file format and a QT4 sample SVG viewer offered the promise of easy development. (The requirements had already specified a preference for QT4.) PostScript was judged to be relatively

complex and was not investigated further. While no alternative should ever be totally discarded PostScript was considered an improbable option.

The SVG support in QT4 was limited to SVG 1.2 Tiny specification (W3C 2008), but this would be sufficient for this type of image.

3.5 First Iteration

The initial SVG image was generated using a simple perl script. Perl was chosen as it is usually installed on Linux machines by default and it can easily be installed on a Microsoft windows system. Although there are many other languages that could have been used, familiarity with this language was a key deciding factor.

In the initial attempt each file was represented by the file name followed by a horizontal coloured bar representing the length of the file. As the width of the resultant graphic was expected to be large, the scale was set so that the size of the smallest file was represented by the width of a single pixel.

Figure 3.5.1 shows the unscaled extreme left of the generated graphic as displayed in a cut down version of the SVG viewer example provided with QT4.

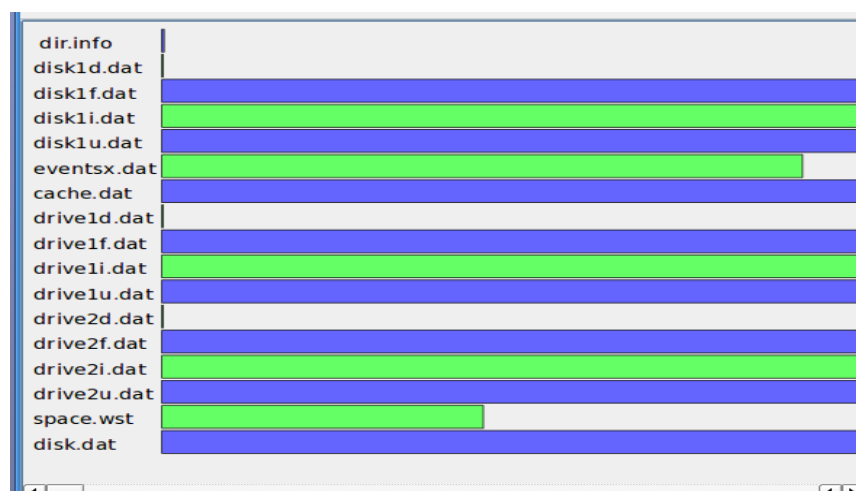


Figure 3.5.1: Initial unscaled graphic

This unscaled version initially looked promising, but when the image was zoomed out to see the extent of the larger files, it was found that the picture rapidly became unreadable as shown by Figure 3.5.2, partially zoomed and Figure 3.5.3, fully zoomed.

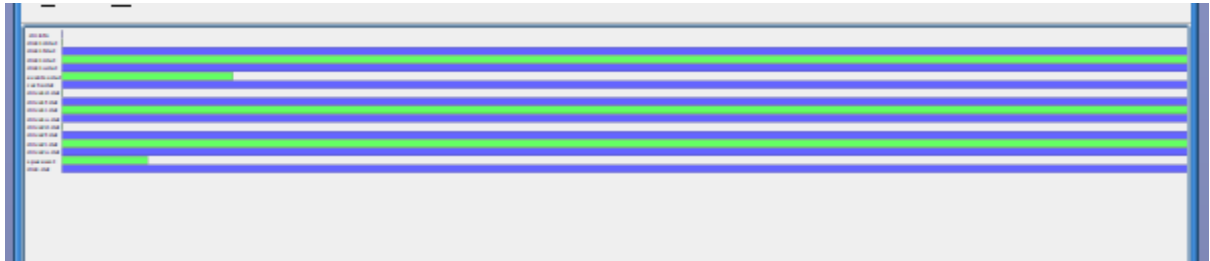


Figure 3.5.2: Partially zoomed

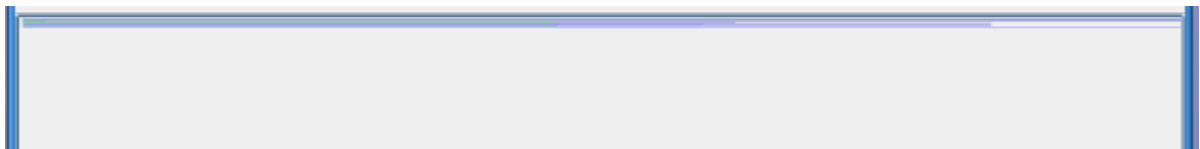


Figure 3.5.3: Fully zoomed

In Figure 3.5.2 the file names are unreadable, but it is possible to see that the small files are much smaller than the big files. Unfortunately as the zoom process continues it becomes harder to see any meaningful information. In Figure 3.5.3 (fully zoomed) it is just possible to discern that the extent of a file size has been reached but it is not possible to determine which file this is.

As the SVG viewer was compiled from open source it was possible to modify the zoom function so that only the width was changed by the zoom process. Figure 3.5.4, where a single axis zoom is used, shows the largest file with other files barely discernible.



Figure 3.5.4: Single axis zoom

Although modifying the zoom process allowed for much better zooming out (to the point where it was possible to see the entire length of the largest file), the resultant image displayed very little if any usable information. The large relative differences meant that the mechanics of the zoom process required considerable user mouse clicks to get from small to large. If the rate of zoom was increased the granularity became such that if similar sized files were to be compared an optimal image zoom may not be available. Although the zoom function in the SVG viewer could be modified to provide a variable rate of zoom, this was not going to provide a novice engineer with an at-a-glance image that could display the file sizes in a meaningful way. That the largest file was very large compared to the smallest file could have easily been determined from the original textual information.

It had been hoped that the produced SVG file would be able to be viewed in a variety of applications, but in order to accommodate the extremes of scale a dedicated application was required. Had this modification produced a usable visualisation, this would have been acceptable even if this restricted future development to a QT solution.

3.6 Non Linear Representations

The first attempt had clearly shown the vast differences in relative file sizes, but had produced little other usable information. Though being able to view this information would be extremely useful to a novice, an experienced engineer would find little, if any, added value in such a revelation.

The second approach to this problem was to consider a non-linear representation of the file sizes. Figure 3.6.1 and Figure 3.6.2 show the file size information scaled using a natural logarithm (base e) and log base-10 respectively. Both logarithms displayed the files in a single view allowing the relative file sizes to be observed. Though there was little to choose, in visual terms, between the representations, it was considered that the log base-10 was preferable. The reason for this choice was not based on any visual improvement but on the fact that interpreting a logarithmic scaled diagram is non-intuitive; that the widely used decibel measuring system was

log base - 10 based and any pre-experience of logarithmic representation was more likely to be drawn from log base -10. The decibel system is widely used to measure audio amplitude and the system is essentially an audio recording device.

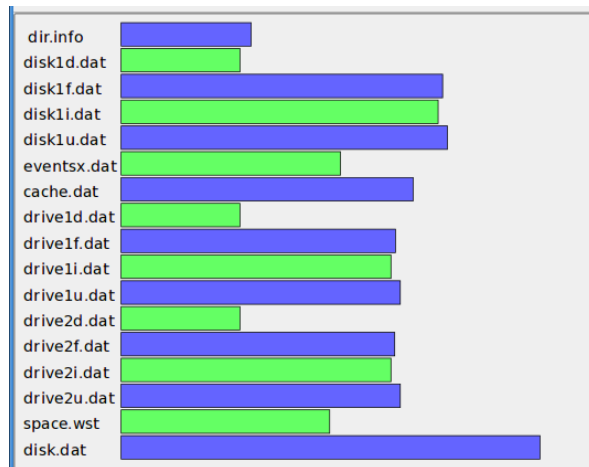


Figure 3.6.1: Log base - e

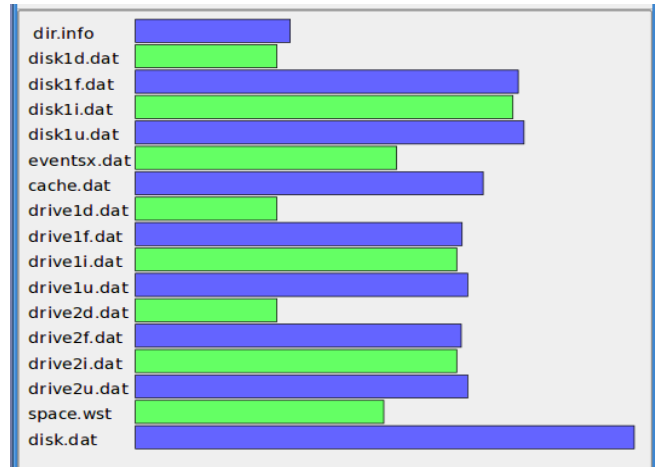


Figure 3.6.2: Log base - 10

Although logarithmic scaling solved all the problems from the initial implementation, it required the observer to either be experienced with this representation or to adjust their perception to use this scale. As this would be the first, and possibly only, representation (of this information) that most users would encounter there would be no “unlearning” required so, even if the image was envisaged as being to scale, it still provided a good overall depiction. With a little practice or with a reference set of sample good images it was anticipated that major problems could be detected from this representation.

3.7 The Most Pertinent Information

Both the two previous iterations had concentrated on representing the relative file sizes for initial assessment without considering what form of assessment would be made. With this at-a-glance representation available it was realised that there may be more useful information to display. One of the key pieces of information derived from the differing file sizes was the number of records each file contained. This information was obtained by dividing the file size by the size of a record of each type of file where the size of the records varies between 4 bytes and 2048 bytes. To

complicate matters further, disk1u.dat's record size is a user defined option. Fortunately this size is known at the time the abs1sr.txt is created and the number of records is calculated making up the final column (Figure 3.3.1 shows a typical abs1sr.txt).

An SVG file was created to depict the number of records per file. Figure 3.7.1, using the number of records, shows the unzoomed left portion of this file and Figure 3.7.2 shows the zoomed-in view of the same information.

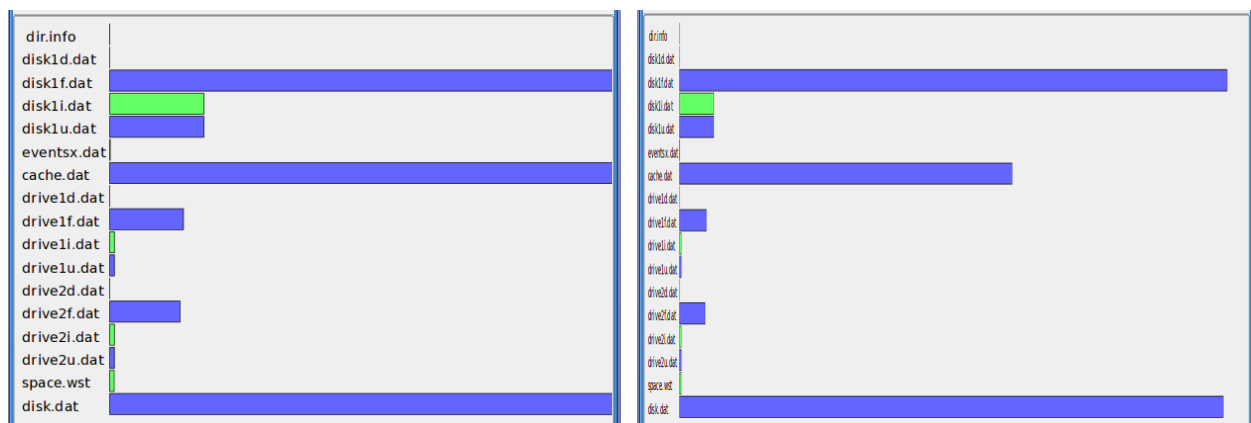


Figure 3.7.1: The number of records unzoomed Figure 3.7.2: The number of records zoomed

The records per file depiction gives a very different perspective. Here it is now easy to see that disk1i.dat and disk1u.dat are about the same size (as indeed are the drive X sets). It can also be seen that disk1f.dat contains the greatest number of records even though it is only about 1/50th of the size of disk.dat.

Even though some useful information can be seen from the graphic, the issue of very big and very small being observed side by side still exists even if the problem is not of the same magnitude. Figure 3.7.3 shows the same information using the log base-10 scale as per the previous iteration.

This image enables some key information to be observed. Files with the same number of records can be seen to be of the same magnitude. Files relating to tape drive information can be seen to be similar for each tape drive present. If such an image were generated for each abs1sr.txt then an engineer would soon become used to the log base-10 scaling and be able to spot large incongruities. Although major discrepancies would be plainly visible in such an image there was still a possibility that significant, but minor differences, may not be so easily discerned.

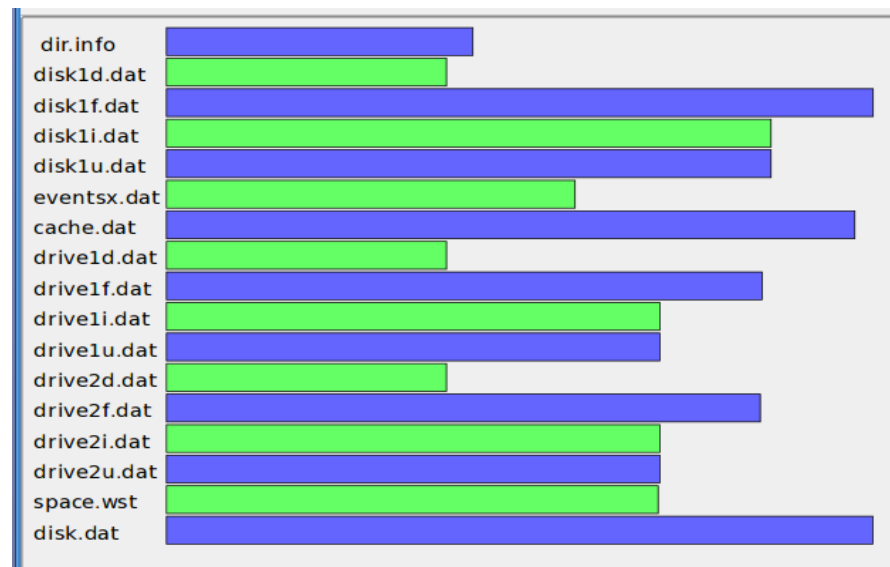


Figure 3.7.3: Number of records in log base-10

3.8 Added Value

Though the logarithmic representation of the records per file was considered to be the most useful, it was felt that there was still some value in the logarithmic representation of file size. However neither graphical representations addressed the potential for fine detail to be overlooked. For example the values 10485760 and 10485768 may not be represented by a visible graphical difference although the textual representation shows the potentially significant discrepancy.

In order to produce a usable product, the graphical information must be intelligently enhanced to highlight the issues for which the engineer searches. The information in an abs1sr.txt file can indicate a number of known problems and has the potential to indicate some not yet encountered. The reason for this is that there are a number of simple mathematical rules to which the files must conform. By checking that a set of files observe the rules any discrepancies can be highlighted.

There was also the possibility of grouping the files into disk and drive clusters, although care would need to be taken not to deviate too far from the familiar textual layout. For example, re-ordering the files could prove to be a big psychological hurdle for an experienced engineer and

potentially mislead a novice.

Figure 3.8.1 shows the repeating groups of d, f, i, and u coloured in a common colour sequence while unique files had their own colours. Here also variation in the separation between files was used to imply grouping.

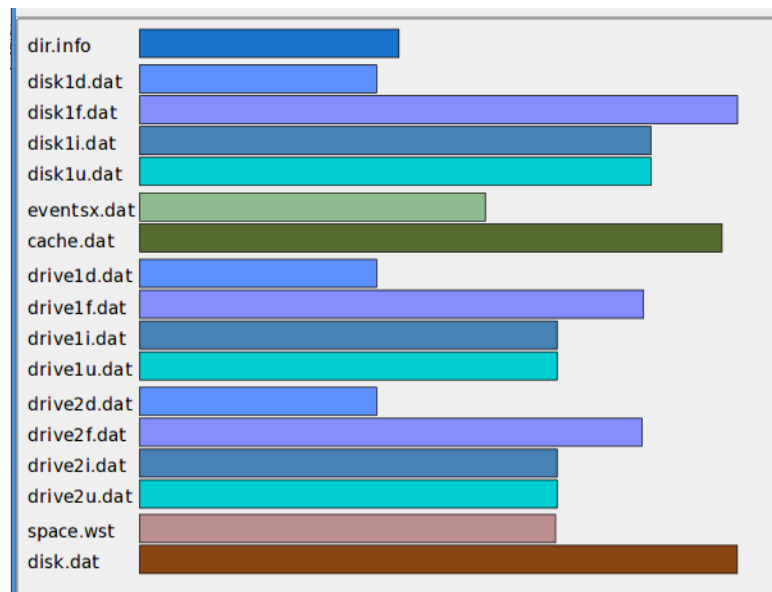


Figure 3.8.1: Grouping files

The choice of colours for this representation proved surprisingly difficult. The colours are defined using a RGB scale (Red Blue and Green) with each colour having possible values between 0 and 255. It had been assumed that if the values of 0, 85, 130 and 255 were used for one colour while the others remained unchanged this would produce four colours on a even colour scale, this was not the case. Whether it was the way the human eye detected colours or the way the computer graphics rendered the colours is not completely understood, but this approach meant that at least two of the resultant colours were barely distinguishable. As a result the colours were chosen by eye, these are not claimed to be the most optimal, but they provide visibly different colours and highlight the repeating file groups satisfying the current requirements. This grouping is further enhanced by small variations in layout.

With a base representation created, the rules to which the file sizes must conform were added. The outcome of these rules were that if a file was determined to be a wrong size then it would be

represented in red; if a file size discrepancy implied that another file was incorrect then the second file would have a red border. Figure 3.8.2 shows the result of this processing.

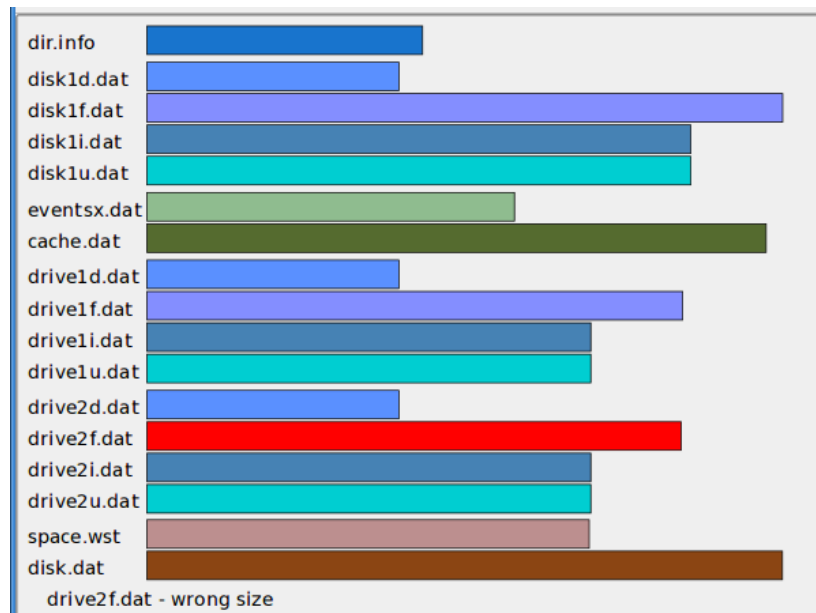


Figure 3.8.2: Adding intelligence

Here it is immediately obvious that drive2f.dat is the wrong size with additional text below indicating the reason for the error even if the bright red colouring does not make this immediately obvious. It would have been possible to present the user with additional information about the extent of the error and suggest correct values, but at this point it was realised that the set of rules used in this analysis was inadequate and that they would not be able to determine all errors. Although the representation was effective, more work was required in calculating the correct sizes and numbers of records for each file.

When this image was viewed it was first considered to be an error in calculation, but a quick check in the pertinent abs1sr.txt file confirmed that this file was indeed the wrong size. It was disconcerting to realise that none of the previous images had highlighted this fact. It was even more disconcerting to realise that the best “picture” of the discrepancy was obtained by knowing it was there and specifically searching for it. Figure 3.8.3 shows a horizontally zoomed-in linear scaled representation of records per file. Although using this representation allows the difference between drive1f.dat and drive2f.dat to be seen, the view does not even permit the identification of

files can just as easily be viewed in Firefox or Opera browsers (tested under Linux KDE X windows and Microsoft Windows XP) and possibly others (untested). That cross-platform browsers can display this file type keeps options open for the Recovery Studio and, while not a current requirement, it does not preclude the possibility of a web enabled application.

With hindsight, the initial visualisation of the files based on size was a little naive. Though the size of a file is important, greater insight can be obtained by considering the number of records. Despite this, the work done working with the file sizes was considered valuable in discounting this approach and in highlighting the need for non-linear representations where file sizes were vastly different. Even if the more sophisticated approach had been taken from the outset this visualisation would need to have been created if only to discount such a method.

The use of a log base-10 scale is considered appropriate in this representation. This does not mean that other non-linear scales can not be used but that this one worked effectively in this instance. It is however disappointing to lose the ability to visually compare small differences between files but, as discovered, a poor and time consuming method of visually comparing files adds no value when there is a textual representation of the same data that can be checked in an instant, especially if the engineer is given an indication to examine this area.

The final iteration's value comes from the additional processing that finds and highlights errors. Although incomplete, if this were integrated into the Recovery Studio it could provide a useful starting point, but the textual information must also be a selectable option if any action is to be taken based on the visualisation.

4 Disk Exploration and Fingerprinting

4.1 Background

By the time the events described in this chapter took place there was a far greater understanding of the recovery processes and the recording system. Although still supervised there was by now greater autonomy and most of the repair decisions were made independently.

Although there was a degree of satisfaction with the previously developed at-a-glance image, it was understood that this technique was only able to cope with a subset of possible errors. Unless some means of determining the correct location values was found the image had only limited application. The search for a means of detecting all faults in the location metadata was still being actively pursued and when this repair came into the laboratory it was considered both a research opportunity and an interesting work problem.

The repair was one in which the tape drive information had been removed. Although a base description of what occurred to the recorder was included in the fault report there was still some question about how the recorder got into this state. The senior engineer had not encountered a machine that had ended up in this state before and it was not certain that the steps the front-line engineer claimed to have taken could have produced a machine in this state.

With no previous instance of this fault there was no known way to repair it. At this time the formula by which the disk was allocated was still not known so the attempt to deduce the missing information by looking at the contents of the disk seemed as good a way to proceed as any other. That it was possible to repair this medium using this method was fortunate. After the repair, the company received new documentation for the process. Businesses like process documentation even if the process is neither likely be required again and if it were it may not always be effective.

The development of the disk fingerprinting technique was purely experimental. Once the first graph was produced the possibilities as a diagnostic tool were immediately obvious and the perl script was put into the repository of diagnostic aids.

4.2 Introduction

The previous chapter considered the information contained in the `dir.info` file and how best to represent this information to an engineer. The key assumption made at this stage was that the information in this file was basically sound and that it could be used to provide a good starting point for the recovery process. This chapter considers a situation in which a recorder has been inappropriately reconfigured and, though the information is consistent for the machine's current state, it no longer reflects the location of the files of the disk and the previously recorded data.

First the unusual situation is described (4.3) and then the practical considerations for this stage of the research are presented (4.4). The detailed description of the techniques used to piece together the contents of the disk are described in 4.5, while the way this repair was tested follows in 4.6. In section 4.7 an attempt to widen the use of the techniques and the pitfalls are documented while the generalisation of the fingerprinting script follows in 4.8. The chapter finishes with section 4.9 and this section's conclusions.

4.3 Missing Information

An initial assessment of a repair revealed that there was no tape drive information in the `dir.info` file when it was known that the recorder in question was fitted with two tape drives. Though incomplete, the recent history of this machine appeared to be that the recorder developed a problem. After several attempts to fix this problem the support engineer physically disconnected the tape drives, deleted the configuration files and then restarted the recorder. Random bursts of corruption found on the recording medium suggested that there may have been SCSI issues but this was not confirmed.

A reasonable question at this point is why did the support engineer take such action? With the

benefit of hindsight this seems a very foolish approach, but it must be remembered that different cultures have differing attitudes to on-site repairs and a non UK company may demand an engineer repair equipment before they are permitted to leave the premises. This repair job did not originate from within the UK. Global differences in the expectations of support engineers is outside the scope of this research.

Although the primary DOS partition was corrupted, it was possible to recover some configuration files which suggested that the recorder had been utilising a user defined record size of 477 bytes. The size of the records in this file is significant, as the size of the file is directly proportional to the size of the records. What was not clear was whether this was the operational value, something generated during the attempted repair or something copied from another machine. This uncertainty meant that very little could be assumed about any of the metadata locations. In this chapter an attempt is made to physically search the disk to discover the true location of the files and to provide the engineer with sufficient information to recreate the broken dir.info.

The objective here was to use the contents of the disk to “reverse engineer” the configuration that was present when the data was created and use this deduced configuration to determine if the current configuration was correct and to recreate the missing information.

4.4 Practical Considerations

While one of the key considerations for this project is the usefulness of graphically representing information for easy dissemination, it must be remembered that a graphical environment is not always possible. The initial investigation into searching this recording medium was done via a command line over an `ssh` connection and using only command line tools and editors such as `dd` and `vi`. The reason for the lack of available GUI was, in this instance, a combination of distance that required a remote connection and security which meant that only limited ports and hence limited services were available. The practical effect of such a working environment meant that all commands or processes had to generate some network traffic to avoid the connection timing out on idle or the command had to be run with the `nohup` prefix and the output monitored with

subsequent logins. (nohup ensures that the process does not terminate if the initiator logs out.)

4.5 Searching the Disk

The damage to the primary partition caused the preprocessing to terminate before an abslsr.txt was created. This failure occurred due to unreadable configuration files. Although it was possible to force this file's creation, it was not considered worthwhile as the information about the current configuration would provide very little assistance in recovering the previous data.

Fortunately an absls.txt was generated. This file contains most of the information of the abslsr.txt, the missing information being the number of records per file. From this file the following information was obtained:

name	bytes	start_address	sectors
dir.info	4096	722925	8
disk1d.dat	2000	722933	4
disk1f.dat	427827712	722937	835601
disk1i.dat	320870400	1558538	626700
disk1u.dat	207228800	2185238	404744
eventsx.dat	1245184	2589982	2432
cache.dat	130023424	2592414	253952
space.wst	57856	2846366	113
disk.dat	217960275968	2846479	425703664

Knowing that dir.info and disk1d.dat should always be 4096 and 2000 bytes in size respectively, and that an initial offset of 722925 sectors was the norm, it was a reasonable assumption that disk1f.dat started at an offset of 722937 sectors from the start of the disk. All information beyond this point was considered likely to be incorrect for restoring this system.

Although reasonably safe to assume, as the absls.txt file was created from examining this portion of the disk, the location of dir.info was checked using the following command:

```
dd if=/dev/sdc skip=722925 bs=512 count=8 | hexdump -Cv | less
```

[dd is a convert and copy program, the parameter "if" refers to the input file in this case it is the

block device `/dev/sdc` (the SCSI RAID) “skip” is the offset into the drive in sectors in this case it is the start of the data partition, “bs” refers to the block size, 512 is the size of a sector and a count of 8 (sectors). The output from `dd` is usually another file or device, but if no destination is given then the default is `stdout`, where in this instance it is piped into `hexdump` and formatted in a Canonical hex+ASCII display which in turn is piped to `less`, a command line text reader.]

The output from this command showed that `dir.info` started in the location expected (see below).

```
00000000  20 64 69 72 2e 69 6e 66 6f 20 00 00 00 10 00 00 | dir.info .....|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 59 | .....Y|
00000020  03 47 64 69 73 6b 31 64 2e 64 61 74 00 00 d0 07 | .Gdiskld.dat..|
00000030  00 00 00 00 00 00 00 10 00 00 00 00 00 00 01 00 | .....|
00000040  04 90 3c 00 64 69 73 6b 31 66 2e 64 61 74 00 00 | ..<.disklf.dat..|
00000050  00 22 80 19 00 00 00 00 00 18 00 00 00 00 00 00 | .".....|
00000060  01 00 04 c0 3b 00 64 69 73 6b 31 69 2e 64 61 74 | ...;.diskli.dat|
```

A similar examination was made of `diskld.dat` and these four sectors of the disk were found to contain only zeros. As the file contains daylight saving information and this feature can be disabled, an empty file is a valid possibility.

The suspected start of `disklf.dat` was examined and found to contain data. While non-conclusive this supported the premise that this was the start of `disklf.dat`. A visual inspection of this data showed “`fe ff ff ff`” to be an often recurring sequence. To discover the extent of this file a shell script was written to check that this sequence recurred regularly.

Sample data from `disklf.dat`:

```
000000c0  1a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
000000d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
000000e0  23 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | #.....|
000000f0  02 00 00 00 fb ff ff ff fc ff ff ff 00 00 00 00 | .....|
00000100  00 00 00 00 00 00 00 00 20 00 00 00 00 00 00 00 | .....|
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000120  00 00 00 00 00 00 00 00 0b 00 00 00 00 00 00 00 | .....|
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000150  00 00 00 00 fe ff ff ff 00 00 00 00 00 00 00 00 | .....|
00000160  00 00 00 00 00 00 00 00 00 00 00 00 fe ff ff ff | .....|
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000180  00 00 00 00 00 00 00 00 fe ff ff ff 00 00 00 00 | .....|
```

The script was a simple loop that outputs the results of a `dd` into `grep` (for “`fe ff ff ff`”) and for each occurrence of the sequence a counter was reset allowing the loop to continue. If the sequence did not occur within an expected scope the script would exit. Given the history of the disk and the presence of bursts of corruption, the script allowed for one sequence to be missed before terminating and reporting the position of the last found sequence. The script was left to run overnight.

The following morning the script was manually aborted. It was clear, that while working, it was impracticably slow and a more efficient method was required. A closer evaluation of the data showed that the sequence “`fb ff ff ff fc ff ff ff`”, while less frequently occurring, was repeated every 1292 bytes and although “`fb ff ff ff`” recurred more often, this was the only occurrence of “`fc ff ff ff`”. The sequence “`fb ff ff ff fc ff ff ff`” was decoded to be the identifiers for a regularly occurring “additional information” block followed by a “special” block. The reason that “`fb ff ff ff`” recurred more often was that additional information blocks may occur as often as required while “special” blocks will only occur every 1292 bytes.

Using this observation the script was then modified to examine only 4 bytes (the size of `fc ff ff ff`) every 1292 bytes and continue only where this was found. This modification improved the estimated run time from a projected execution time of 3.5 days to a recorded time of 36 minutes.

This performance improvement meant that the script would halt if any corruption was encountered, but with the significant decrease in running time it became realistic to continually monitor its progress rather than leave it running over a weekend where a false stop would be problematic. It also meant that there was virtually no likelihood of another file having the same sequence in the same location and providing a false positive. The end of `disk1f.dat` was found to conform to the `abs1.txt` prediction. While unexpected, this coincidence gave some confidence in the results.

It was anticipated that `disk1i.dat` would be next file to be examined. The following command was used to examine this portion of the disk.


```
dd if=/dev/sdc skip=1558538 bs=512 count=1 | hexdump -Cv | less
```

This produced the following output:

```
00000000  01 74 d7 93 77 7e 02 00  ff ff ff ff ff ff ff ff  |.t0.w~..00000000|
00000010  93 5f ec 44 00 00 00 00  ff ff ff ff 00 08 14 00  |._0D....0000....|
00000020  06 06 00 00 34 9e 00 00  10 0d 00 00 00 00 00 00  |....4.....|
00000030  02 74 d7 93 77 7e 02 00  00 74 d7 93 77 7e 02 00  |.t0.w~..t0.w~..|
00000040  a7 5f ec 44 00 00 00 00  ff ff ff ff 00 08 07 00  |0_0D....0000....|
00000050  06 06 00 00 f4 42 00 00  34 ae 00 00 00 00 00 00  |....0B..40.....|
00000060  03 74 d7 93 77 7e 02 00  01 74 d7 93 77 7e 02 00  |.t0.w~..t0.w~..|
00000070  b4 5f ec 44 00 00 00 00  ff ff ff ff 00 08 0a 00  |0_0D....0000....|
00000080  06 06 00 00 a0 52 00 00  b8 fe 00 00 00 00 00 00  |....0R..00.....|
00000090  04 74 d7 93 77 7e 02 00  02 74 d7 93 77 7e 02 00  |.t0.w~..t0.w~..|
000000a0  be 5f ec 44 00 00 00 00  ff ff ff ff 00 08 3e 03  |0_0D....0000..>|
000000b0  06 06 00 00 fc 56 19 00  a0 52 01 00 00 00 00 00  |....0V..0R.....|
000000c0  05 74 d7 93 77 7e 02 00  03 74 d7 93 77 7e 02 00  |.t0.w~..t0.w~..|
000000d0  fc 62 ec 44 00 00 00 00  ff ff ff ff 00 08 0d 00  |0b0D....0000....|
```

This hexdump showed that this file contained a regular pattern repeating every 48 bytes. As each record in this file was 48 bytes in size this was the expected repeat interval. The pattern, while distinct, evolved over the records. This evolution was explained by the fact that this structure contained incremental counters and date/time representations.

For this recorder the sequence “77 7e 02 00” was the high end representation of the recorder identity and this was used as part of the record numbering sequence. Ideally this sequence would be found repeated twice every third line, as linked list next sequence number and previous number form part of the record. Since calls have a start and end (the absence of previous or next segments are both represented by -1) then it should be found at least once on every third line. Here the output from `dd` was piped to `hexdump` and then to `grep`, the output of which was itself piped to `wc -l`. This counted the number of lines containing this sequence and when used with a block size of 512 at least 10 lines should be expected to be counted. However this is not always the case as a caller may change their mind and put the phone down before it is answered or there may be a glitch on the line creating a phantom call. After some experimenting, a value of 4 lines per sector was considered a reasonable, robust minimum value to allow for such occurrences.

The script was run and it stopped after about 200,000 sectors. This portion of the disk was examined and there appeared to be a large area of corruption. The address of this anomaly was recorded and the script was restarted a sector later on the disk.

The script stopped a second time a little before the offset predicted by the `absls.txt`. This time an examination of the disk revealed empty space. The most likely explanation of the empty space was that this was an as yet unused portion of the `disk1i.dat`'s allocation. This hypothesis could be supported by finding the next area of disk that was used and examining its contents.

The script was modified to search for all zeros and abort at the next area of data. Data was next found at an offset of 2185238 and a visual inspection suggested that this was `disk1u.dat`.

```
dd if=/dev/sdb skip=2185238 bs=512 count=1500 | hexdump -Cv | less
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 36     | .....6|
00000020  36 30 38 00 00 38 39 39 30 00 00 00 00 00 00 00 | 608..8990.....|
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000040  00 00 00 49 6e 00 00 30 00 00 00 00 00 00 00 00 | ...In..0.....|
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
00000090  00 00 00 00 00 00 00 00 38 39 39 30 00 00 00 00 | .....8990.....|
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
000000b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
```

The `disk1u.dat` files contain the information fields that the customer has chosen to record. These may be telephone numbers, operator identity, whether the call is incoming or outgoing or any other information the customer required. The size of a user defined field (udf) depends on how the recording system has been configured and cannot be predicted, but the record size information may be deduced from other files if this information is lost when a recorder fails. An examination of this (suspected) `disk1u.dat` file showed strings of what appeared to be telephone numbers. This type of data was consistent with presumed file type even though very little use appeared to have been made of this user defined feature. Despite being suspected that a record size of 477 was used there was too little recorded information in the file to determine if this was correct; additionally there were many records in which no data was present.

The script examining this portion of the disk searched for between 24 and 31 lines of empty space per sector and allowed for four consecutive sectors to fall outside these tests before aborting. Despite allowing a number of sectors to fail the tests, the script frequently stopped for false alarms making this part of the disk exploration slow and tedious. These false alarms consisted of

bursts of corruption and sequences of empty sectors. At an offset of 7987133 no more udf like data was found and the next portion of the disk contained all zeros.

The empty space terminated at an offset of 8413069 sectors. If the udf field size were 477 bytes then this would be the expected end of disk1u.dat. It was considered that a partially written disk1u.dat (with a record size of 477 and with empty space for further records) to be the most probable explanation of the disk at this stage; this was further supported by the fact that disk1i.dat had also finished before its predicted end and both these files should use their space at the same percentage rate. Although there was little confidence in the previous values in abs1s.txt, some reassurance was gained when coincidental values were obtained experimentally. However at this point the abs1s.txt value for the size of the disk1u.dat was clearly wrong. For this value to be correct either the udf size was 31 bytes, which seemed suspiciously small, or it contained differing numbers of records to disk1i.dat, which should never occur.

At an offset of 8413070 was a file presumed to be eventx.dat. This file was very different to the previous files in that it contained densely packed information, but without any obvious repeating pattern. Given that the eventx.dat file is an audit log of the recorder these are not surprising. This file was found to give way to empty space at an offset of 8415502.

```
dd if=/dev/sdb skip=8413070 bs=512 count=1500 | hexdump -Cv | less
00000000  84 38 ea 46 b0 01 fd 01 01 00 01 b3 08 03 00 2d | .80F0.0....0...-|
00000010  00 00 00 84 38 ea 46 b0 01 00 12 03 00 01 2b 00 | ...80F0.....+..|
00000020  00 00 00 00 00 00 84 38 ea 46 b0 01 fd 0a 03 00 | .....80F0.0...|
00000030  01 f3 fe ff ff 00 00 00 00 84 38 ea 46 b0 01 01 | .0000.....80F0..|
00000040  91 09 00 0a f2 13 00 00 01 00 00 00 84 38 ea 46 | .....0.....80F|
00000050  b0 01 01 91 09 00 0a f2 13 00 00 02 00 00 00 84 | 0.....0.....|
00000060  38 ea 46 b0 01 01 91 09 00 0a f2 13 00 00 03 00 | 80F0.....0.....|
00000070  00 00 84 38 ea 46 b0 01 01 91 09 00 0a f2 13 00 | ...80F0.....0..|
00000080  00 01 00 00 00 84 38 ea 46 b0 01 01 91 09 00 0a | .....80F0.....|
00000090  f2 13 00 00 04 00 00 00 84 38 ea 46 b0 01 01 91 | 0.....80F0...|
000000a0  09 00 0a f2 13 00 00 05 00 00 00 84 38 ea 46 b0 | ...0.....80F0|
000000b0  01 01 91 09 00 0a f2 13 00 00 01 00 00 00 84 38 | .....0.....8|
000000c0  ea 46 e0 79 00 63 05 00 01 00 00 00 00 00 00 00 | 0F0y.c.....|
000000d0  00 37 39 ea 46 b0 01 fd 01 01 00 01 b3 08 03 00 | .790F0.0....0...|
000000e0  2d 00 00 00 37 39 ea 46 b0 01 00 12 03 00 01 2c | -...790F0.....|
000000f0  00 00 00 00 00 00 00 00 37 39 ea 46 b0 01 fd 01 01 | .....790F0.0..|
```

An eventx.dat file is expected to be 1245184 bytes in size. The extent of this file exactly matched this size without any trailing empty space. To eliminate any possibility of this being a

coincidence the data on this portion of the disk was copied into a file and the data decoded as if it were an eventx.dat file. It was reassuring to find that audit information was present in this file as predicted and it was also observed that the records at the start of the file followed immediately after those at the end. This indicated that the file had filled its allocated region and began overwriting the oldest data. Since the records were contiguous, both the start and end of this file must be correct. With the location of eventx.dat confirmed there could be far greater confidence in the size and type of all the previous files derived experimentally.

After this point the disk exploration became less clear cut. Empty space was found to occupy offset 8415502 to 8669458. This could indicate cache.dat, drive1d.dat or both. Without knowing what was found here, no further mapping could be made for sure. The rest of the disk was explored and more areas of data were found. This data was similar to the data found in FAT, index and udf files and was presumed to be tape related data. This suspicion was based on the assumption that the next area of the disk contained tape drive 1 and 2 files.

The following table summarises what was found from the end of the eventx.dat file:

Type	Start	Finish
Empty space	8415502	8669458
Fat like	8669458	8702938
Empty space	8702938	8751378
Fat like	8751378	8768460
Empty space	8768460	8812818
Fat like	8812818	8831014
Index like	8831014	8835510
Fat like	8835510	8841276
Index like	8841276	8851630
Empty space	8851630	9423382
Fat like	9423382	9498205
Empty space	9498205	9505302
Fat like	9505302	9520546
Empty space	9520546	9566742
Fat like	9566742	9605058

Udf like	9605058	9635285
empty space	9635285	10178579
disk.dat	10178579	

It must be noted that a change from one data type to another may indicate a change of usage. A change from some form of data to empty space may indicate the end of a file or the maximum usage of the file.

On its own, the layout of this portion of the disk was uncertain. The next step was to look at the expected files and their permitted sizes and try to find an explanation for the observed disk structures. Some of the file sizes are fixed and others depend on the type of tape drive fitted (of which two types exist, DDS3 and DDS4). The following table shows the expected files and, where known, their possible sizes in bytes.

File	Static size	DDS3 size	DDS4 size
cache.dat	130023424		
drive1d.dat	2000		
drive1f.dat		25165824	41943040
drive1i.dat		18874368	31457280
drive1u.dat		187564032 *	312606720 *
drive2d.dat	2000		
drive2f.dat		25165824	41943040
drive2i.dat		18874368	31457280
drive2u.dat		187564032 *	312606720 *
space.wst			
disk.dat			

The values marked with a * are calculated assuming a udf field size of 477 which, given the coincidence of the experimentally determined size and the predicted size of disk1u.dat using this figure, was considered the most likely value.

Trying to make any correlation between the DDS3 file sizes and the observed data failed. When the DDS4 file sizes were used a number of boundary matches were found **if** it was assumed that

the empty data at the start was both cache.dat and drive1d.dat.

The following table shows the information collected at this stage of the research and theoretical values for the unconfirmed files using the default values for DDS4 drives. The values marked with * are those that are to some degree confirmed by a physical change of data type observed on the disk.

Name	Size in bytes	Start address in sectors	Number of whole sectors
dir.info	4096	722925 *	8
disk1d.dat	2000	722933 *	4
disk1f.dat	427827712	722937 *	835601
disk1i.dat	320870400	1558538 *	626700
disk1u.dat	3188649600	2185238 *	46227832
eventsx.dat	1245184	8413070 *	2432
cache.dat	130023424	8415502	253952
drive1d.dat	2000	8669454	4
drive1f.dat	41943040	8669458 *	81920
drive1i.dat	31457280	8751378 *	61440
drive1u.dat	312606720	8812818 *	610560
drive2d.dat	2000	9423378	4
drive2f.dat	41943040	9423382 *	81920
drive2i.dat	31457280	9505302 *	61440
drive2u.dat	312606720	9566742 *	610560
space.wst	653824	10177302	1277
disk.dat	214206241280	10178579 *	

The only unconfirmed boundaries were those that occurred in empty space. As both cache.dat and drive1d.dat appeared to be empty there was no way to determine where one started and the other ended, as a result the start of drive1d.dat was established on the basis of its known size. A similar situation occurred where drive2u.dat was considered to be part used and space.wst considered as empty data. With these exceptions, the disk exploration and the use of known possible values provided a very good match between theoretical values and observed values.

Of the empty files on the disk, cache.dat was orally confirmed as being a legacy file and not in use at the time, hence it was expected to be empty space. That the disk and drive d.dat files were empty due to daylight savings being turned off was unconfirmed but plausible. It was also possible that the recorder had not been in service long enough to encounter a change, while space.wst (as its name suggests) was known to be used as a filler to ensure the space used by disk.dat contains an integral number of records.

4.6 Putting the Theoretical Values to the Test.

To ensure that the values derived from this experiment were correct, the dir.info file was recreated with these values, and written back to the correct portion of the disk. By using the corrected state and configuration files it was possible to check if the recorded data could be replayed correctly.

As far as it was possible existing copies of the original files were taken from the DOS partition. Although some attempt had been made to take backup copies of the critical files, this was done after some key values had been reset. Despite this the remaining information was still useful in recreating basic configuration files.

Generating and copying back the new dir.info was the first step. After this step it was possible to use an automated script to collect textual representations of the current state of the disk files. Unlike the previous attempt where the information was unavailable as the wrong locations of the disk were being used to extract data, this attempt appeared to complete successfully and provide credible data. This data was used to calculate the missing values for the configuration files.

As the DOS partition was corrupted a known good image of a similar DOS partition was used to overwrite this partition. The DOS file system could have been formatted and reinstalled, but, as a good image was available this was considered an acceptable short-cut. The recovered configuration files were modified so that the pointers to the system files were back to the values

prior to the failure and then copied back into the file system of the corrected DOS partition.

The RAID, which held the repaired data, was disconnected from the recovery PC and connected to a recorder with hardware compatible with the original. The system was then powered on. When it had finished booting a selection of calls were replayed. The replaying of the calls provided a high degree of confidence that the operation had been successful. In fact there was audible human speech, the predicted call length matched the actual replayed time and there was no garbling or sudden switching between unrelated calls. Tapes were then inserted into the recorder and the previously unrecorded data was backed up to tape - these were the deliverables of this recovery task. Prior to dispatch, further tests were carried out to prove the integrity of the tapes (and hence the critical portion of the recovered disk). The tests found the tapes to be acceptable.

4.7 Wider Considerations

On this occasion the disk analysis, combined with known information, was able to recreate the missing dir.info structure. Subsequent to this disk exploration it was confirmed that the tape drives had been DDS4. Key to the success of this approach was the fact that this disk was fairly clean. Clean in this context meant that there had been no previous use of the disk and that space that was not currently in use was “empty”, containing only zeros.

Disks or RAID systems may be transferred between recorder systems, a disk that was previously in a machine with DDS3 tapes may end up in a system with DDS4 or vice versa. Udf field sizes may be changed possibly giving rise to two sets of file beginnings for the tape drives and, more importantly, more than one starting point for disk.dat.

If the starting point for disk.dat is assumed to be too early then the offset of the information from the (true) beginning of the file will be incorrect. The presence of no information at or before the normal offset should alert an observant engineer, but if disk.dat is assumed to start after its real start then information from the start of this file may be overlooked. In the case of systems being

reset, the data to be recovered is usually at the start of this file.

An attempt was made to automate the disk crawling scripts for this system and, although it was possible to automatically identify `dir.info`, `disk1d.dat`, `disk1f.dat` and the start of `disk1i.dat`, areas of corruption repeatedly caused the script to fail. The areas of corruption appeared to be raw audio information written to random portions of the disk. These areas could last for several sectors or just a few dozen bytes. Given that the size of these areas of corruption could be greater than a `disk1d.dat` or `driveXd.dat` files meant that any program that ignored short lived corruption risked missing a key file. Worse still was the fact that various subroutines had been optimised for efficiency for each type of file, thus if a file was missed there was a reasonable chance that the wrong matching criteria would be used for the wrong file types. Such corruption was always a possibility in the systems received for recovery.

Although the use of disk crawling scripts and human intervention had been fundamental to the recovery of this particular disk, this did not necessarily mean that the method was universally useful. It was disappointing to realise that this method was not going to provide a quick fix. Before completely abandoning this method it was decided to adapt the script to crawl a disk, attempting to fingerprint the sectors and see what if any value could be extracted from the results.

4.8 Disk Fingerprinting

The disk crawling script was rewritten to `dd` a sector to `hexdump` (as before) but this time the `-dv` switches were used to format the output in a decimal format with no ASCII as below.

```
0000000 11427 32896 51907 00001 65535 65535 65535 65535
0000010 23440 16276 01024 00106 65535 65535 00064 01259
0000020 02561 00000 51360 00036 08532 65030 00002 00000
0000030 11428 32896 51907 00001 65535 65535 65535 65535
0000040 23440 16276 01024 00118 65535 65535 00064 01259
0000050 02561 00000 51468 00036 57648 65029 00002 00000
0000060 11429 32896 51907 00001 65535 65535 65535 65535
0000070 23440 16276 01024 00120 65535 65535 00064 01259
0000080 02561 00000 51576 00036 41264 65029 00002 00000
0000090 11430 32896 51907 00001 65535 65535 65535 65535
00000a0 23440 16276 01024 00123 65535 65535 00064 01261
```

```
00000b0  02561  00000  55832  00036  24880  65029  00002  00000
00000c0  11431  32896  51907  00001  11423  32896  51907  00001
```

The fingerprint was designated to be the sum of all the decimal values for a sector. It was recognised that a few large values could produce a similar output to many smaller values, but it was decided to use this technique in order to understand what, if anything, could be determined.

The output from the script was two numbers per sector, the sector offset and the sum for that sector. It was recognised that this would create a very large output file and that the process could take days to complete. (Although this was a disadvantage, large output files and slow disk traversal are common features of many data recovery techniques.) When the script had traversed the disk from offset 722925 (the start of dir.info) to beyond the start of disk.dat (offset 11810163 sectors) it was stopped. Beyond this point there would only be a continuation of disk.dat so it was not expected to reveal anything new. This data took about a day and a half to collect and was 152M in size. The processing of the data was done using gnuplot, this command line driven scientific graph generator took less than a minute. The result for this system is shown in the following diagrams, the first Figure 4.8.1 is the raw graph and the second Figure 4.8.2 is edited to assist interpretation. The X axis is the offset in sectors and the Y axis is the sum of the bytes (as described above) of the sector.

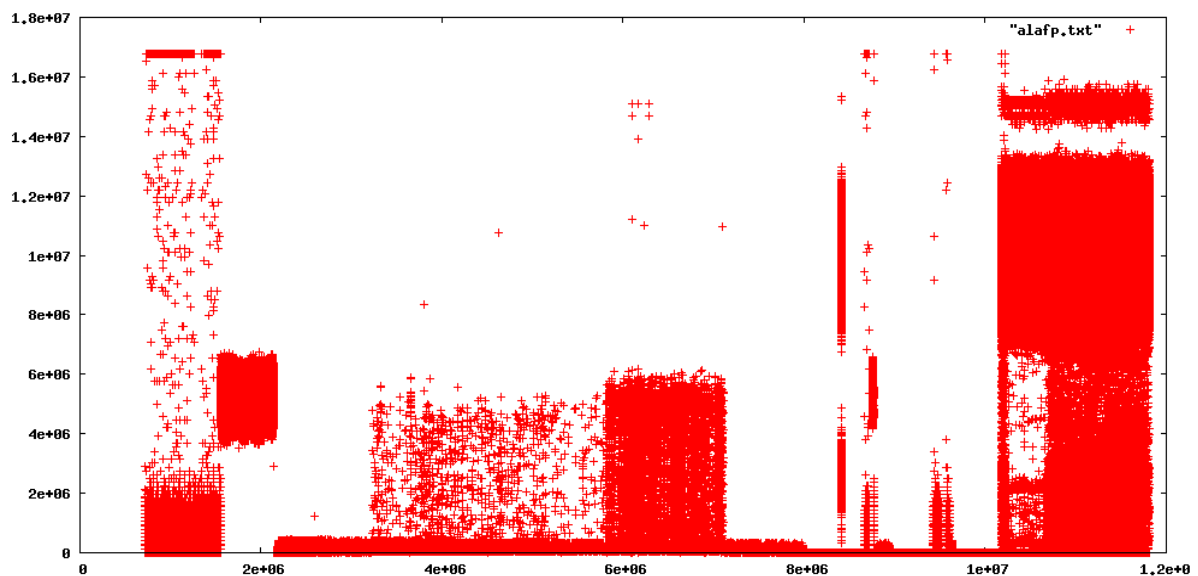


Figure 4.8.1: Raw graph

It should be noted that the empty left portion of the graphs is the application partition that was not iterated and that the empty space to the right was where the script terminated. The method used to “fingerprint” the disk sectors did produce visibly different patterns for each of the file types. This result was unexpected and potentially very useful.

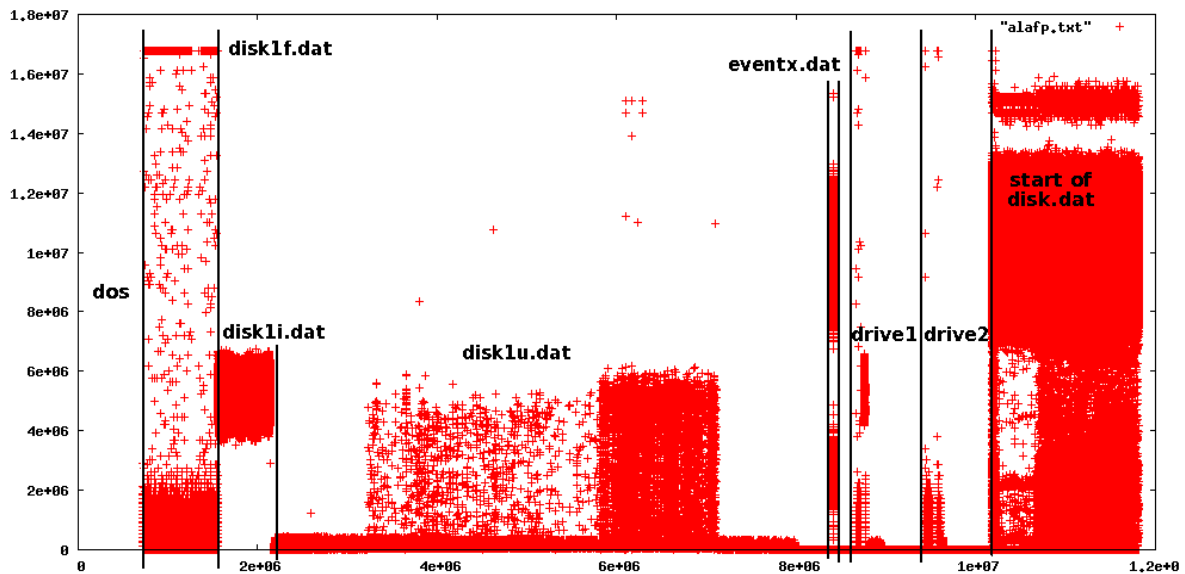


Figure 4.8.2: Annotated graph

From this graph it is possible to identify most of the main files although, as with previous representations, the small files such as dir.info and disk1d.dat are indiscernible. Despite this there are some very interesting patterns visible, in disk1f.dat nearly all sector fingerprints are either less than 2000000 or about 17000000; for disk1i.dat the data lies between (approximately) 4000000 and 7000000. The disk1u.dat file is especially interesting as it shows a customer barely using the user defined information, then increasing the usage twice before reverting back to minimal usage, however there can be nothing in this data that can explain the rationale or policies behind such a history. Just as the human assisted disk crawling produced a clearly defined eventx.dat, with some less than obvious definitions for the tape drive related files, the fingerprinting technique was similarly effective.

Disk fingerprinting produced a potentially useful visualisation, but before discussing its value, a second scenario is considered. In this instance there was no question about the layout of the files on the disk, this fingerprint was taken of the next system to be recovered and was intended to verify the findings from the previous disk exploration. Again the two following graphs show both

raw (Figure 4.8.3) and interpreted information (Figure 4.8.4), the greyed areas in the second annotated graph shows data from a previous configuration.

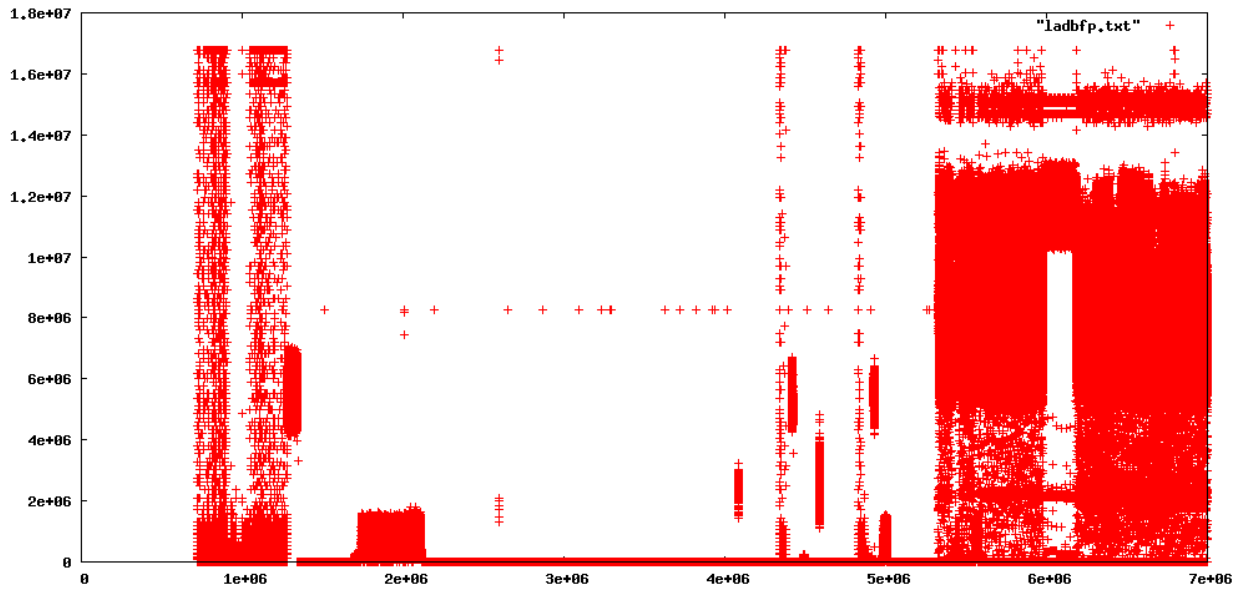


Figure 4.8.3: Second disk - raw

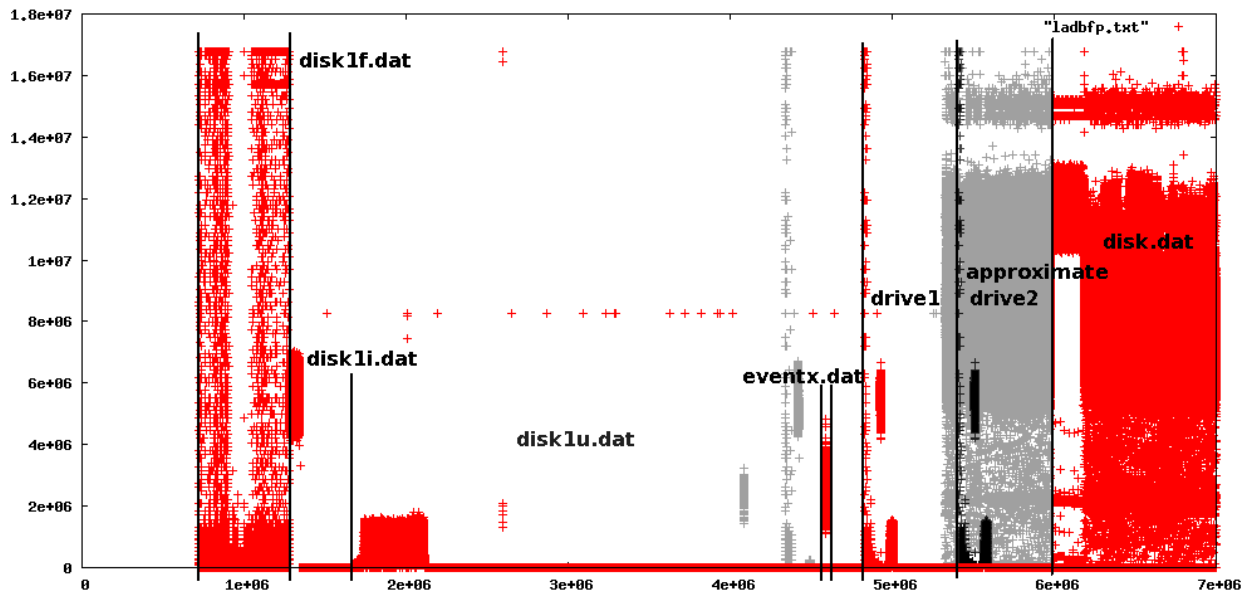


Figure 4.8.4: Second disk - annotated

At first glance these graphs show the same pattern as the previous drive except that this one is more clearly defined. This graph is however a composite of two different usage patterns. In the

earliest usage the user defined fields are relatively small and take up a smaller portion of the disk causing the old eventx.dat, tape drive files and disk.dat file to be written at smaller offsets than the current usage. In time it is expected that the disk1u.dat file would overwrite the old eventx.dat and drive1 files, although it is not certain whether the current tape drive files would ever completely erase the old start of disk.dat.

This previous usage was not known at the time of this repair; its presence was irrelevant to the recovery process as in this instance the layout of the disk was known and proved to be correct.

4.9 Chapter Conclusions

In the case of missing information about the location of files on the disk a physical examination of the disk contents can assist in finding these files. When a system has been recording on a clean disk, this disk exploration may produce definitive answers although some caution is required in interpreting empty areas of disk. As a technique for reliably recreating or verifying the location metadata, it is not suitable for all instances and as such cannot be considered the solution to discovering the underlying location metadata.

Specific file type crawling scripts can be useful, however these are prone to fail in the presence of corruption on the disk. Also if the scripts are programmed to ignore corruption they may also miss valid files. An experienced human eye viewing hexdumps of sectors can still be one of the best methods of determining the content at any point on the disk.

Disk fingerprinting can produce very a clear picture of what is present on a disk, but this does not distinguish between current and previous configurations. As a result it may provide misleading information. The process can take several days to complete depending on the speed of the processor and the disk or RAID configuration. While this delay is not in itself prohibitive, an experienced engineer would be able to selectively view portions of the disk and identify key boundaries in a few hours. That the fingerprinting is able to produce a very good visualisation of the state of the disk may yet prove to be useful as a diagnostic aid. Where possible further

fingerprints will be taken to see if any particular characteristics, for a full system or single file type, are associated with a particular type of failure.

This chapter has shown that it is possible to recreate information missing from dir.info files by intelligently exploring the disk and that the disk fingerprinting technique can produce very good visualisation of the state of the disk even if this visualisation has the potential to mislead.

5 Calculating the Correct File Sizes

5.1 Background

By the time this stage of the research was attempted a reasonable degree of confidence had been gained in the specialist data recovery techniques used in the repair processes. Despite working on many recovery tasks each new repair was still a puzzle and no two repairs had been the same.

Attempting to find a way to detect all metadata location errors was still an active pursuit. When access to the recording application's source code was granted it was assumed that finding the solution would be a relatively simple task. The was not the case. As this chapter illustrates, it required several months to derive the formula. The research attempt that led to the solution was based on knowledge acquired from understanding the source code and the types of methods employed (in the code). Such knowledge formed the basis of plausible conjectures made in relation to the formula used by the application to apportion the medium.

Once the medium allocation formula was found it was easy to enhance the at-a-glance image generation to produce far better error detection. At this stage it was possible to detect all errors and offer the correct solution. This would provide a significant step toward the development of the Recovery studio.

Coincidental to the formula being (re)discovered was a recovery job in which the location metadata had been changed in a way not previously believed possible. It is still believed that the formula could recalculate the correct location information, but it was the disk fingerprinting process that gave insights into the recording problem.

5.2 Introduction

The previous two chapters have explored the representation of the systems' files with an at-a-

glance image and, when the full extent of the possible errors was realised, exploring the possibility of determining the correct configuration from the state of the disk. However neither of the techniques tested to this point produced a reliable method of determining the correct layout and sizes of the files. In this chapter a definitive method of calculating these critical system parameters is pursued and the results are used with the previously developed visualisation.

This chapter briefly describes the assumption that the source code would be easy to unravel in section 5.3. Some explanation for this is given in section 5.4, while 5.5 describes the formula. However, although the formula matches all available examples, the chance to use it to reconstruct a blatantly wrong location allocation fails as the data to verify the reconstruction is missing (5.8). As a demonstration of potential, the graphic produced in Chapter 3 is represented using the additional information available (5.9).

5.3 The Code

The source code for the recorder's application was made available and it was initially assumed that it would be possible to extract the formula used to allocated the file sizes. This assumption was incorrect. Even with advice and notes (from a previous attempt) the search through the code took over a month before any meaningful information could be extracted. The information that was extracted was not the formula but the type of techniques used in manipulating variables.

Whilst it would be easy to write a long critical passage on the ugliness of the source code, its historical context must be considered. The code was developed over several years and by many developers with more than one company buyout. Its development is currently reduced to the point where only operationally important bug fixes are considered. While this was not observed first hand, there are stories of minor simple changes in one area of the code causing catastrophic failures in other modules with no obvious reason ever found. However, it is suspected that some critical timing may be involved.

Some comments within the code appear to have been superseded by other changes leaving tracts

of documentation at best misleading and at worst completely wrong. Yet despite the potential disinformation many useful facts were obtainable even if all information had to be otherwise validated.

One key observation that was readily available was the frequent use of a rounding up function that took two arguments (X , Y). The function added the value of X with the value of $Y-1$ and then performed an integer division on this sum using Y as the divisor. The result of this division was then multiplied by Y . This meant that the function returned the next integer above X that was exactly divisible by Y . That such a function existed and was frequently used enabled conjectures to be made and tested while attempting to find a formula that fitted all the collected examples.

5.4 The Files and their Calculations

The recorder application's source code made references to the size of the data storage partition and the number of tape drives. Although not completely understood, the code had some methods related to the type of tape drives used. Using insights gained from the code and information already known to the recovery engineers the basic inputs were identified. These inputs were:

1. The size of the partition.
2. The number of tape drives.
3. The type of tape drives.
4. The size of the user defined fields.

The size of the recording medium can be determined using the `sg3_utils sg_readcap` (Gilbert D 2009) utility which returns (among other things) the number of bytes capacity; for example, the output for this PC is shown in Figure 5.4.1. During a normal initial scan of a repair, this data is collected to a readcap file so that the information is available for further processing (e.g., by the future Recovery Studio) or for human reading.

As the size of the operating system partition is a constant size, subtracting this (370137600 bytes)

will leave the size of the data partition.

```
[root@smudge ch5v2]# sg_readcap /dev/sda
Read Capacity results:
  Last logical block address=312581807 (0x12a19eaf), Number of blocks=312581808
  Logical block length=512 bytes
Hence:
  Device size: 160041885696 bytes, 152627.8 MiB, 160.04 GB
[root@smudge ch5v2]#
```

Figure 5.4.1: Output from sg_readcap

The number and type of tape drives and the user defined field size should all be extractable from the configuration files, but it is possible that if the application partition is damaged then these configuration files may be inaccessible. If this information is not available from the files, a visual inspection of the recorder would reveal the number and type of tape drives and the size of the user defined fields should be known to the company and their front line support.

Although, in theory, other settings could be changed and this would have an effect on the location metadata, there are no known systems where these changes have been made nor is it anticipated that these configuration options would ever be adjusted from their default values. It is the possibility that such anomalies may legitimately exist. This means that full automation of a repair is unlikely to be possible and that a degree of human verification will always be required.

Some information is obvious and some information can be deduced, for example all the files have a start address, a size, a number of records and the size of a record. The size will be the exact product of the record size and the number of records. All files start on a sector boundary even if the previous file occupies only a few bytes of the preceding sector. Some files are constant in size, some are dependent on the size of the disk and some on the type of tape drives fitted. The number of tape drives fitted can determine whether or not some files exist while most files are always present.

To examine the information available for each file:

name	information
dir.info	The file information is stored in dir.info, which is always located at an offset of 722925 sectors from the start of the disk (or RAID file system). This file is

	always 4096 bytes in size and it occupies exactly 8 sectors. It may contain information for the files associated with up to eight tape drives though in practice most systems will have two or less of these devices fitted.
disk1d.dat	This file starts after dir.info at 722933 sectors (the start of dir.info + the size of dir.info in sectors). This file is a constant 2000 bytes in size and while occupying 3.90625 sectors this is rounded up to 4 sectors when considering the start of the next file. The record size of this file is 1 byte and there are 2000 records.
disk1f.dat	The start of this file is always 722937 (the start of disk1d.dat + 4 sectors). This file is calculated from the size of the hard drive (or RAID). Although the code that performs this operation has not been isolated, the formula appears to be that the number of records is the result of the size of the disk in bytes divided by 2048 (2K) rounded up to the next number divisible by 323. As there are 4 bytes to a record, the file size is 4 times the number of records. There is however a minimum size of 13107340 records (52429360 bytes). According to comments in the code this is approximately 25G ($1024 * 1024 * 25 = 26214400$), however it can be seen that the real value is twice this (at the time of this research, the point where this value is doubled had not been found and this information is derived from observed data taken from successful repairs).
disk1i.dat	The number of records in this file is the size of the disk in bytes divided by $(2048 * 16)$ rounded down to a whole integer. As there are 48 bytes to a record the file size is the number of records multiplied by 48. As with disk1f.dat there is a minimum number of records of 819200 (size of 39321600 bytes). The start point of this file is dependent on the size of disk1f.dat (in turn dependent on the size of the disk).
disk1u.dat	This file is dependent on the user defined data. There are the same number of records as disk1i.dat and the size is that of $(\text{disk1i.dat} / 48) * \text{udf size}$. The start of the file is dependent on the sizes of disk1f.dat and disk1i.dat (in turn decided by the size of the disk).
eventsx.dat	The start of this file is determined both by user configurable data and the size of the disk. This file has a constant size of 1245184 bytes. It always contains

	65536 records and occupies exactly 2432 sectors.	
cache.dat	This starts 2432 sectors after eventsx.dat. The file is also a constant 130023424 bytes with one byte per record. It occupies 253952 sectors.	
The files relating to the tape drives typically come as a pair though there are configurations where only one or no tape drives are present. Where tape drives exist the type of drive determines the file sizes. The following two sets of files are possible. Here X is the drive number and the maximum number of drives is 8.		
	DDS3	DDS4
driveXd.dat	2000	2000
driveXf.dat	25165824	41943040
driveXi.dat	18874368	31457280
driveXu.dat	$(18874368 / 48) * \text{udf}$	$(31457280 / 48) * \text{udf}$
space.wst	The remainder of the disk is as many whole records of disk.dat as possible with any unused space being taken up by space.wst. So the size of space.wst is the modulus of the remaining space in bytes divided by 661504 (the size of a record of disk.dat in bytes).	
disk.dat	This file occupies the remainder of the disk.	


5.5 Testing

The above formulae were verified by looking at previous repairs and comparing the calculated values for abslsr.txt files with the actual retrieved values. Where differences existed the observed data was found to be suspect and in many cases this was the underlying reason for the recovery task. Whilst it is not proven that all eventualities have been taken into account, at the time of this research, the formula proved correct for all existing historical data. If the assumption is that this formula is true in all cases then no instance has yet been found that disproves this assumption.

5.6 Using the Information – Design and Implementation

There are many ways in which the solution could be presented to a user from a command line program using arguments to a fully blown interactive graphical application. While no future development direction has been discounted an initial proof-of-concept application has been developed as a pseudo web server mostly following HTTP 1.0 standards (Berners-Lee et al 1996) written in perl. This permits information entry and returned results to be displayed in any browser from any operating system; allows more than one query to be handled from more than one location simultaneously and the browser provides a familiar interface. As there was only one instance of the application running, any development changes could be deployed instantly by editing the source, stopping and restarting the server.

The following images Figure 5.6.1 and Figure 5.6.2 show the data input page, with data, and the returned results.



The screenshot shows a web browser window titled "http://localhost:8889/ - Opera". The browser's address bar and navigation buttons are visible. The main content area displays a form with the heading "Enter information". The form contains four input fields with the following labels and values:

Enter partition size in bytes ..	219048322048
Enter total udf size	273
Enter DDS Type (3 or 4)	4
Enter Number of drives	2

Below the input fields is a button labeled "Work it out".

Figure 5.6.1: Data input web page

In order to calculate the file information the size of the disk is required; the number and type of the tape drives fitted to the recording system and the size of the user defined fields. The data input page uses a simple HTML (Hyper Text Mark-up Language) form to input the data which when submitted is used to calculate the predicted abslsr.txt that is returned to the browser to be compared with the data in the actual abslsr.txt generated during the initial scan of the recording media.

The screenshot shows a web browser window with the title "abslsr information". The browser's address bar contains the URL "http://localhost:8889/info?n1=219048322048&n2=273&n3=4&n4=2". The main content area displays a table with five columns: name, bytes, start_address, sectors, and records. The table lists various files and their corresponding metadata values.

name	bytes	start_address	sectors	records
dir.info	4096	722925	8	4096
disk1d.dat	2000	722933	4	2000
disk1f.dat	427829004	722937	835604	106957251
disk1i.dat	320871552	1558541	626703	6684824
disk1u.dat	1824956952	2185244	3564370	6684824
eventx.dat	1245184	5749614	2432	65536
cache.dat	130023424	5752046	253952	130023424
drive1d.dat	2000	6005998	4	2000
drive1f.dat	41943040	6006002	81920	10485760
drive1i.dat	31457280	6087922	61440	655360
drive1u.dat	178913280	6149362	349440	655360
drive2d.dat	2000	6498802	4	2000
drive2f.dat	41943040	6498806	81920	10485760
drive2i.dat	31457280	6580726	61440	655360
drive2u.dat	178913280	6642166	349440	655360
space.wst	586240	6991606	1145	586240
disk.dat	215838171136	6992751	421558928	105389732

Figure 5.6.2: Metadata predicted layout

5.7 Using the Information as Part of the Repair Process

The web server was left running on a PC in the laboratory. When a repair came in and the initial scan was carried out the resultant abslsr.txt file was observed; while previously its contents may have been presumed to be correct, it was now possible to point a browser at this web server and generate the predicted metadata layout. If the two sets of information were in agreement then the abslsr.txt and the metadata locations on the disk could be assumed to be correct and the repair could proceed with a high degree of confidence in this data that may otherwise only be obtained gradually as the repair progressed. This was a significant improvement to the initial assessment.

When the comparison failed, an early warning of a problem could be given. Apparently plausible

data could be detected as being suspect and an underlying cause determined. With such information possibly hours of a data recovery engineer's time could be saved by the realisation that errors were present. In the case of missing data, as in the previous chapter, the information could be easily recreated without time consuming hours spent searching the data partition.

If corruption occurred in the location metadata area and all location data was lost then the information could be recreated from scratch, a feat not previously possible. Even though such a situation may be very rare this would previously have taken many days to correct. It should be remembered that each repair is unique and nothing could have predicted the state in which the following repair was found.

5.8 An Example Beyond Repair

Not all disks can be repaired. The following `abslsr.txt` was generated from a recorder that was unable to replay a specific call.

name	bytes	start_address	sectors	records
<code>dir.info</code>	4096	722925	8	4096
<code>diskld.dat</code>	2000	722933	4	2000
<code>disklf.dat</code>	427829004	722937	835604	106957251
<code>oops.dat</code>	217631682560	1558541	425061880	unknown
<code>diskli.dat</code>	320871552	426620421	626703	6684824
<code>disklu.dat</code>	13416441768	427247124	26203988	6684824
<code>eventsx.dat</code>	1245184	453451112	2432	65536
<code>cache.dat</code>	130023424	453453544	253952	130023424
<code>space.wst</code>	611840	453707496	1195	611840
<code>disk.dat</code>	-12880390144	453708691	-25157011	-6289253

The problem with this disk is that an extra file, `oops.dat`, had been generated and this distorted the entire disk layout. The file `oops.dat` is a legacy repair file that was generated in specific circumstances for a specific customer; its scarcity meant that it had not been previously mentioned and even where it occurs the total sectors used by `oops.dat` and `disklf.dat` is equivalent to a normal `disklf.dat`. (This is a very good example of local knowledge being able to override the predicted data, but still make use of its contents.) `Oops.dat` should not be present on this disk and up until this point no one had even speculated what would happen if someone tried to modify

the media layout in this fashion where this fix was not required. As previously observed, it is never wise to predict the actions of front line support.

The presence of negative numbers in disk.dat attributes is caused by the difference between the calculated space used and the size of the disk in this case, due to the inappropriate oops.dat, the difference is negative. Despite the fact that all the parameters for calculating the correct location metadata are believed to be known, no usable index, FAT, event or udf information could be processed from the disk. The reason for this was that although the location of the files was believed to be correct, the data was overwritten by audio data when the application started writing to the beginning of the partition and then would not write metadata in the audio data area.

As no further progress could be achieved on that day, a disk fingerprint (as described in the previous chapter) was initiated and left to run overnight. The resultant image Figure 5.8.1 shows the start of the data partition and gives an insight into what was occurring on the disk while it was operational.

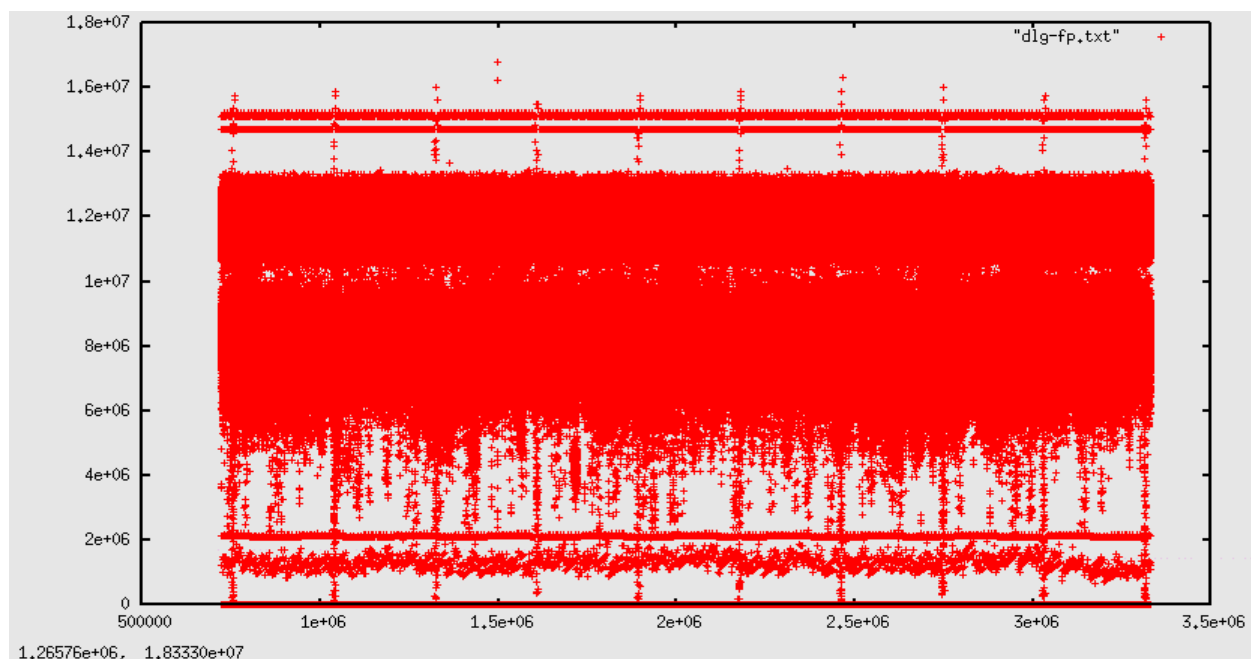


Figure 5.8.1: The start of the partition where the metadata should be.

The image reveals that the start of this partition was occupied by disk.dat data (in fact the whole of the data partition would be subsequently found to contain disk.dat data). Although this may

seem a reasonable outcome for such an unusual abslsr layout, it breaks a lot of the assumptions currently existing in this area of data recovery. The disk fingerprint for this area showed some interesting patterns, the most obvious of which is the repeating vertical lines every X sectors.

Despite the fact that there were no index or FAT files, it was still considered possible that a recovery may have been effected by extracting embedded “backup” copies of the key metadata files that were encoded into disk.dat. At this point it was believed that by relocating the data to a larger drive and using these synthetic metadata files, of size and location determined by the uncovered formula, it would be possible to replay a reasonable percentage of the audio. To be able to repair a disk in this fashion would prove that the concept of recreating abslsr data from the formula as a viable repair technique.

The above scenario proved to be extremely optimistic. When the synthetic indexes were created they were discovered to be less than 20% usable with large portions of the data completely barren of these metadata structures. As the available data tended to occur in bursts, these bursts were examined to see if there was a usable string of continuity. It was discovered that these bursts were 20 minutes apart. 20 minutes is the time interval after which the tape, if present, will record any unrecorded data from the disk so there was some speculation as to the possible mechanism that was causing the index information to be written to disk only at this time (although no satisfactory full explanation has yet been found). It was then that the correlation was made between the regular lines in the fingerprint and the presence of index information. It was not the separation of these lines by sectors that was important, but the separation by time combined with the fact that these patterns occurred coincidentally with the presence of (backup) index metadata in the main data.

The image from the misplaced disk.dat, while recognisable as disk.dat data, shows some differences with previous images created from more usual disk.dat data. If it is compared with a normal disk.dat where backup index data is present such as Figure 5.8.2, it can be seen that there is continuous pattern of this data and it is reasonable to assume that the difference in pattern is caused by the presence of this information stored within the data. The backup information may be only index information, but as no tests were done to see if any other backup data was present, it would be unsafe to assume that the lack of index information is the whole explanation of the

visible differences between the two disk captures. The reason the presence of other data was not tested was that this work was performed in an operational environment and one in which research must not be allowed to interrupt the core business. There was insufficient index information to recover the audio irrespective of whether other information was available or not. Other repairs were waiting and there was no advantage to the company in searching for this data.

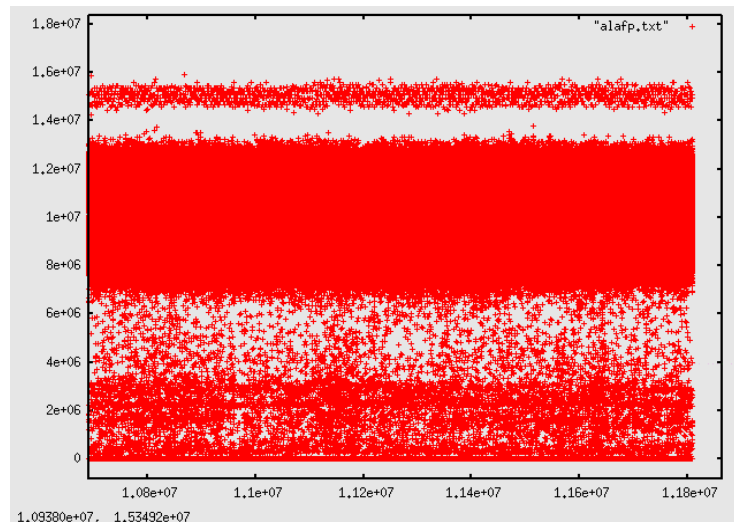


Figure 5.8.2: A portion of a normal disk.dat

Despite the uncertainty about exactly what happened while the recorder was running in this unlikely configuration, there is evidence that the fingerprint process can potentially identify not only types of recorded data, but also patterns that could be symptomatic of certain fault conditions.

A hypothesis that the metadata could be successfully recreated from scratch in this instance has been neither proved nor disproved. It is believed that calculated values for this repair are correct, but without any metadata available the audio cannot be reconstructed to see if it replays recorded conversations and thus provide a verifiable answer.

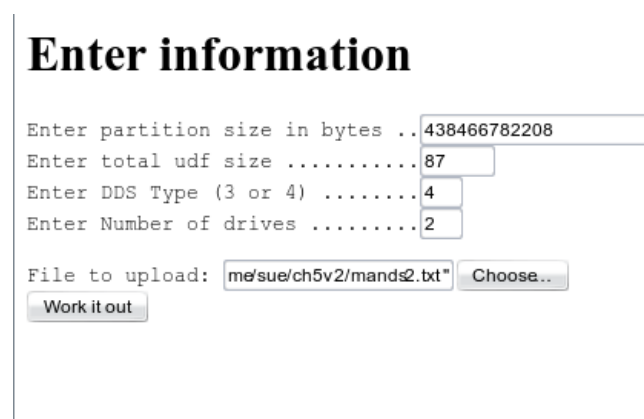
Even though the previous chapter decided that the disk fingerprinting could not provide a definitive method of predicting the location of the metadata, the technique provided useful insights into what happened to the recording and is the subject of the next chapter.

5.9 One Final Consideration

Chapter 3 focused on presenting the information to a novice engineer in an at-a-glance graphic. Although the recovery studio is a long way from being built, it is possible to use the formula discovered in this chapter to improve the generated image so that additional errors can be detected.

The discovery of the underlying formula permitted a far greater number of errors to be detected and graphically displayed, therefore the SVG generation software was added into the web server. As both applications had been written using perl the integration was relatively straight-forward, but with the underlying formula available, it was possible to detect and highlight a greater number of errors. The data entry page was modified so that an abs1sr.txt could be uploaded and analysed.

There were a few issues in the development that required some HTTP 1.1 (Fielding et al. 1999) commands being used to prevent Opera Version 9.62 for Linux from caching images. These problems were circumnavigated and the resultant screen shots were captured to show a working prototype. Disappointingly Firefox would not display the images in-line although it would show them if the option to open an image were selected. The following images show the data input page, Figure 5.9.1 and a good location metadata file results (Figure 5.9.2).



The screenshot shows a web form with the following fields and values:

Field Label	Value
Enter partition size in bytes ..	438466782208
Enter total udf size	87
Enter DDS Type (3 or 4)	4
Enter Number of drives	2

Below the input fields, there is a 'File to upload:' label, a text box containing 'me/sue/ch5v2/mands2.txt', and a 'Choose..' button. At the bottom left of the form is a 'Work it out' button.

Figure 5.9.1: Data input form

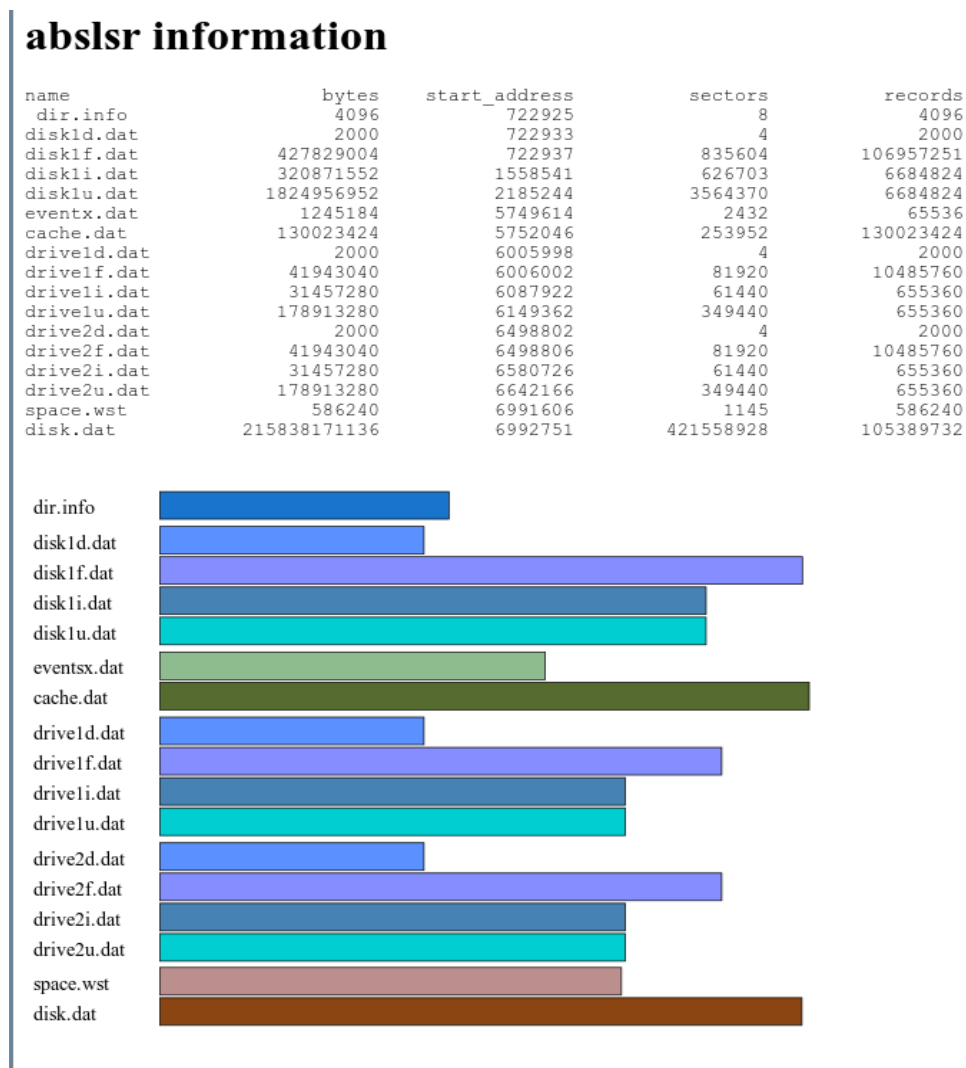


Figure 5.9.2: A good abslsr.txt file

The above graphic shows the data input. If no file is uploaded (the upload box is left blank) then the results are as before generating only a correct set of values as per Figure 5.6.2. If a correct abslsr.txt is uploaded then the above is a typical good output.

Although this image is now part of a web page it is fundamentally the same as the one created in Chapter 4. The additional benefit realised consists of being able to detect the wrong size of file as well as the cascading effect on other files (as shown in Figure 5.9.3). This is the same input data used in Chapter 3 with a much clearer indication of the effect of the problem is being displayed.

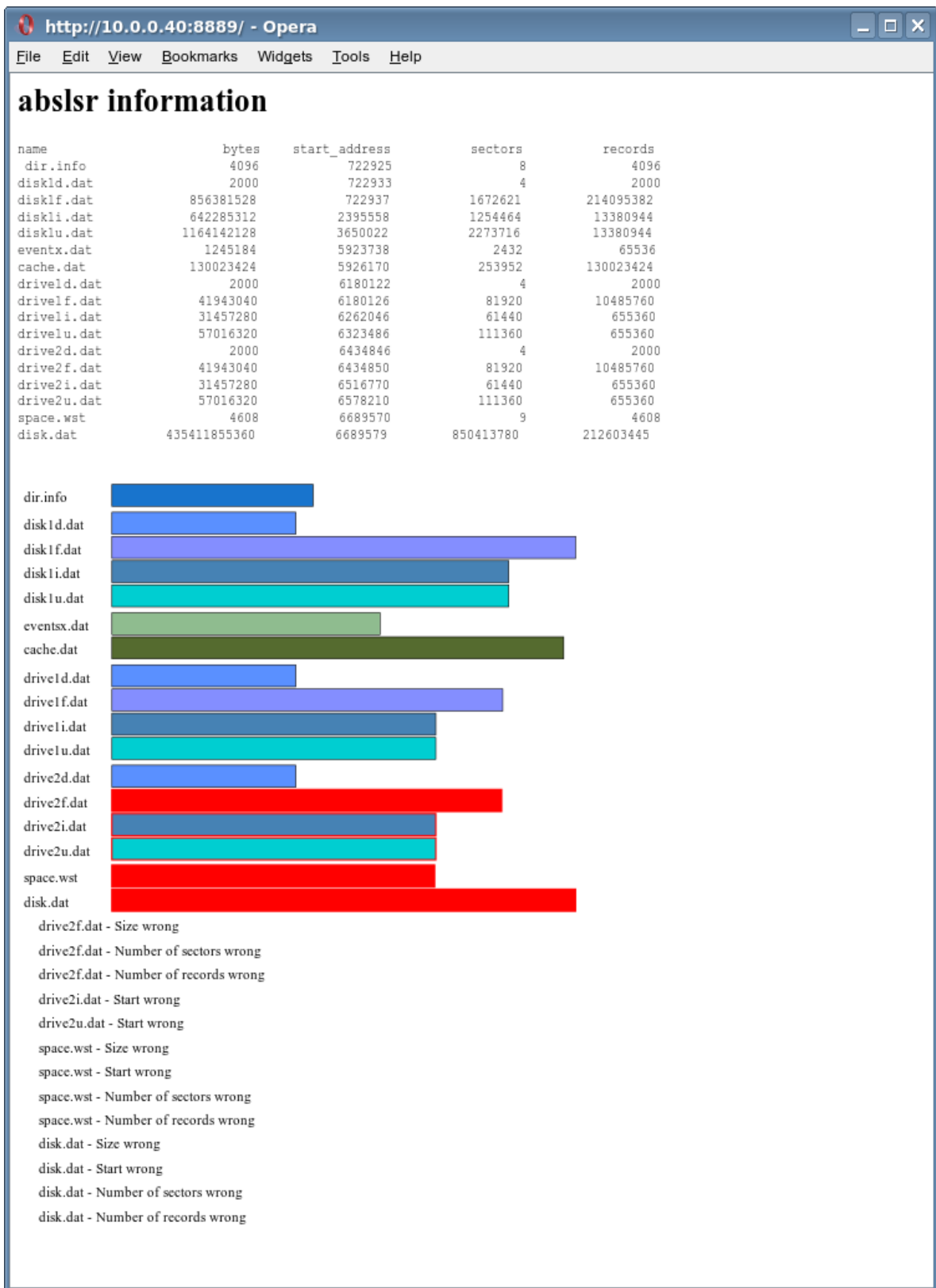


Figure 5.9.3: The corrected layout with abslsr.txt errors depicted in red below

The previous diagram shows that the `drive2f.dat` is of incorrect size, but unlike the image in Chapter 3 this one is able to identify not only that the size is wrong, but also that the sectors occupied and the number of records are incorrect. This sets the border of the incorrect file to red as well as the fill. As the subsequent files are out of place their borders are also red although the files themselves are of the correct size (the fill colour is correct). As the sizes of `space.wst` and `disk.dat` are derived from the remaining space in the partition they too are incorrect. The lower portion of the image is made up of textual information and gives a list of discrepancies encountered, that the first error reported is likely to be the root cause of the dislocation is no coincidence.

Although for an expert working in the laboratory the availability of correct information which can be compared with the actual information may be more useful, for a novice in the field the at-a-glance image can convey the underlying problem in a easy to understand fashion, thus supporting the concept that a graphical representation of the location metadata can be used to give a diagnosis of the root cause of the problem.

For both the expert and the novice sufficient information exists in the textual results to create a text file that can be converted into binary data that may be written to the `dir.info` region of the disk to correct the problem.

5.10 Chapter Summary

In this chapter the formula for determining the location metadata was deduced after access was gained to the source code. Although it was not possible to determine this formula directly from the code it was possible to gain an insight into the type of methods used in apportioning the recording media and with that knowledge use an informed process of trial and error that led to the creation of a formula for predicting the correct layout of metadata and data.

While it is still possible that this formula may not hold true for all instances of the system, so far

no instance has been encountered that does not conform. After a considerable amount of testing sufficient confidence was gained and a mini web server was built to provide an interface to the formula in the recovery laboratory. The web server application was used to verify subsequent repair location metadata and when a repair was encountered where the location information was clearly wrong it was used to predict the correct values. Unfortunately it was not possible to test these predicted values as the damage was such that the metadata was either never written to disk or was overwritten by audio data. Interestingly it was the disk fingerprinting technique developed in the previous chapter that gave the greatest insight into the nature of the recording before the fault had been discovered. As the primary objective for this research was to find a reliable method of verifying the location metadata it had not been reasonable to change direction and explore the potential for this technique, but now that the location metadata formula had been identified it would be possible to revisit the disk fingerprinting process and experiment in this area. (See next chapter, Disk Fingerprinting – An Exploration.)

The formula was integrated into the heuristic analysis of the abs1sr.txt that was used to generate an at-a-glance image which in turn was imported into the mini web server. The result was a proof-of-concept server application that could be used with a web browser that supported SVG graphics. As the design and development of the Recovery Studio cannot be predicted, further refinement would be of little benefit.

6 Disk Fingerprinting – An Exploration

6.1 Background

Shortly after the formula by which the recording medium was allocated, and this part of the research was concluded, the employment was terminated.

Despite the work done in designing the at-a-glance representation, it is expected that these concepts have been stored until the development of the Recovery Studio begins. It is also expected that the senior engineer has already extracted the formula used to allocate the recording medium and that this has either been rewritten in the python programming language or has been incorporated into the initial assessment shell script. This will provide immediate benefit with minimal development overhead. There is a certain satisfaction in knowing that every current repair has one less area of uncertainty.

Whilst the previous stages of the research had been done in a commercial laboratory this next stage was done from home. Here it was possible to focus on the disk fingerprinting experiments without the overhead of company or customer issues. The freedom from these issues was offset by the fact that all data collected and documents written were stored in a single (geographical) location. To mitigate the risk of data loss through, for example, a house fire or burglary, an additional 500GB drive was purchased and a backup of all data was stored off-site.

The drives used for this part of the research were all privately owned and though some of the software installations were old or broken all these drives were physically functional.

6.2 Introduction.

With the formula rediscovered the research could be considered complete, but this chapter returns

to the disk fingerprinting technique to see if it has wider possibilities and takes a look at the information available from iterating a collection of hard drives. There is nothing special in these drives or their contents; the novelty comes from this technique providing a new way of seeing the data. As each of these drives were in computers of various states of repair there are subtle differences in the way the script was run and each machine has a description of the method used. Section 6.3 gives a brief outline of common methods.

The first disk is part of an old Microsoft Windows 95B system. As the disk was a moderate 10 Gigabytes in capacity it could be traversed in a relatively short period of time. This drive was used to compare the differences between different partitions and to see the consequences of formatting a logical drive. This disk is described in greater detail in section 6.4 with specific comparisons between two C: drives described in 6.5. This chapter also includes before and after comparisons of a simple format command (6.6).

In section 6.7 an examination is made of a Linux disk while 6.8 compares two fresh installations, identical except for the choice to use volume encryption. In section 6.9 similarities in the way data is stored on a hard drive are noted across different file systems before the chapter is concluded in section 6.10.

6.3 Common Methods

For logistical reasons the computers used were powered down each night. This meant that the fingerprinting process was only running for about half of the day and traversing a disk could take months. Each morning the process was restarted from the point reached the day before and each evening the scripts were amended to set the skip value to one less than the sector reached. (The skip value is an argument given to the Unix `dd` process and is the offset into the disk in sectors.) This meant that the resulting text files had two values in common with the previous file, although this prevented the files being simply concatenated it allowed a basic sanity check to ensure that there were no gaps or overlaps.

It would be easy to modify the perl script to take command line arguments, but editing the starting point each time the script was run produced a list of stages reached during each working day. This history allowed some predictions to be made about the duration of a particular disk traversal and indicated in which file a particular portion of data was located.

All data was plotted using gnuplot. Each section below describes the exact commands used with an explanation of why different options were chosen. For all graphs the X axis was the offset in sectors from the start of the disk and the Y axis the count for that sector.

6.4 Disk 1 General

The first disk was in a Pentium II machine with a speed of 450 MHz, 128 Megabytes of memory and a hard drive of a nominal 10 Gigabytes. The PC was set to boot from the CD drive. A Live Fedora 8 KDE CD was used as a temporary operating system, this should have booted to the KDE X windows system, but the video card was not able to cope and only the command line was available. As the command line was all that was required this was sufficient.

The PC also had a second hard drive that was not fingerprinted, but this was used to store the output data. On a Microsoft Windows machine a primary partition is by default identified as the C: drive, unusually, this first drive was partitioned with 3 primary partitions (effectively 3 C: drives) and an extended partition containing 3 logical drives. Only one of the primary partitions could be active at any time so the other two had to be hidden partitions (Sammes, Jenkinson 2007).

Three C: drives is unusual and this machine was originally partitioned this way so that more than one installation of Windows 95B could be installed. Exact records about how this installation was achieved have not been kept, but it is believed that an old version of PowerQuest's Partition Magic (now owned by Symantec) (Symantec Corporation 2009) was used to layout the drive partitions.

The original reasoning behind this unusual partitioning was to install Microsoft Windows 95B on the first and second primary partitions and leave the third spare for a possible trial of the forthcoming Microsoft Windows 98. With the two partitions installed with the same operating system one was used for everyday working, whilst the second was used to test out perceived risky operations, such as new video drivers or major system configuration changes. If these were successful then the main drive was similarly upgraded. Microsoft Windows 95B did not have features such as system restore or check pointing a stable configuration so reinstalling a system was more common than with later Microsoft products.

The working Windows 95B installation did fail, but recovery was trivial as all that was required was to change the active partition and reboot. There has always been an intention to repair or reinstall the failed installation, however it was left in its failed state so that any data found to be missing could be copied to another drive. It had been this way for many years and by now it is considered that any useful information has either been salvaged or completely forgotten that its presence is irrelevant.

The Redhat Fedora fdisk (util-linux-ng 2.13) using the -l option reported the following for this disk.

```
Disk /dev/sda: 10.1 GB, 10141286400 bytes
255 heads, 63 sectors/track, 1232 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x00000000
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1		1	213	1710891	16	Hidden FAT16
/dev/sda2	*	214	426	1710922+	6	FAT16
/dev/sda3		427	621	1566337+	16	Hidden FAT16
/dev/sda4		622	1232	4907857+	f	W95 Ext'd (LBA)
/dev/sda5		622	830	1678761	6	FAT16
/dev/sda6		831	1025	1566306	6	FAT16
/dev/sda7		1026	1232	1662696	6	FAT16

Partition Magic, the application that originally created the layout, provides this information:

```
C1: hidden fat 16
First physical Sector 63
Last physical Sector 3,421,844
total sectors 3,421,782
1,751,952,384 bytes
52% used 910,896,136
```

48% free 841,056,256

C2: active fat 06

First physical Sector 3,421,845
 last physical Sector 6,843,689
 total sectors 3,421,845
 1,751,984,640 bytes
 78% used 1,361,848,832
 22% free 390,135,808

C3: hidden fat 16

First physical Sector 6,843,690
 last physical Sector 9,976,364
 total 3,132,675
 1,603,929,600 bytes
 0% used 263,680
 100% free 1,603,665,920

Extended partition 0F

First physical Sector 9,976,365
 last physical Sector 19,792,079
 total 9,815,715

E-95 USABLES - 06 fat 16

First physical Sector 9,976,428
 last physical Sector 13,333,949
 total 3,357,522
 1,719,051,264 bytes
 91% used 1,561,568,256
 9% free 157,483,008

F:FACTS - 06 fat 16

First physical Sector 13,334,013
 last physical Sector 16,466,624
 total 3,132,612
 1,603,897,344 bytes
 62% used 992,446,464
 38% free 611,450,880

G:MASTERS - 06 fat 16

First physical Sector 16,466,688
 last physical Sector 19,792,079
 total 3,325,392
 1,702,600,704 bytes
 58% used 982,458,368
 42% free 720,142,336

This information is summarised in Figure 6.4.1. It should be noted that the absence of a D: drive is due to the second hard drive having a primary partition and this partition taking the “D” letter.

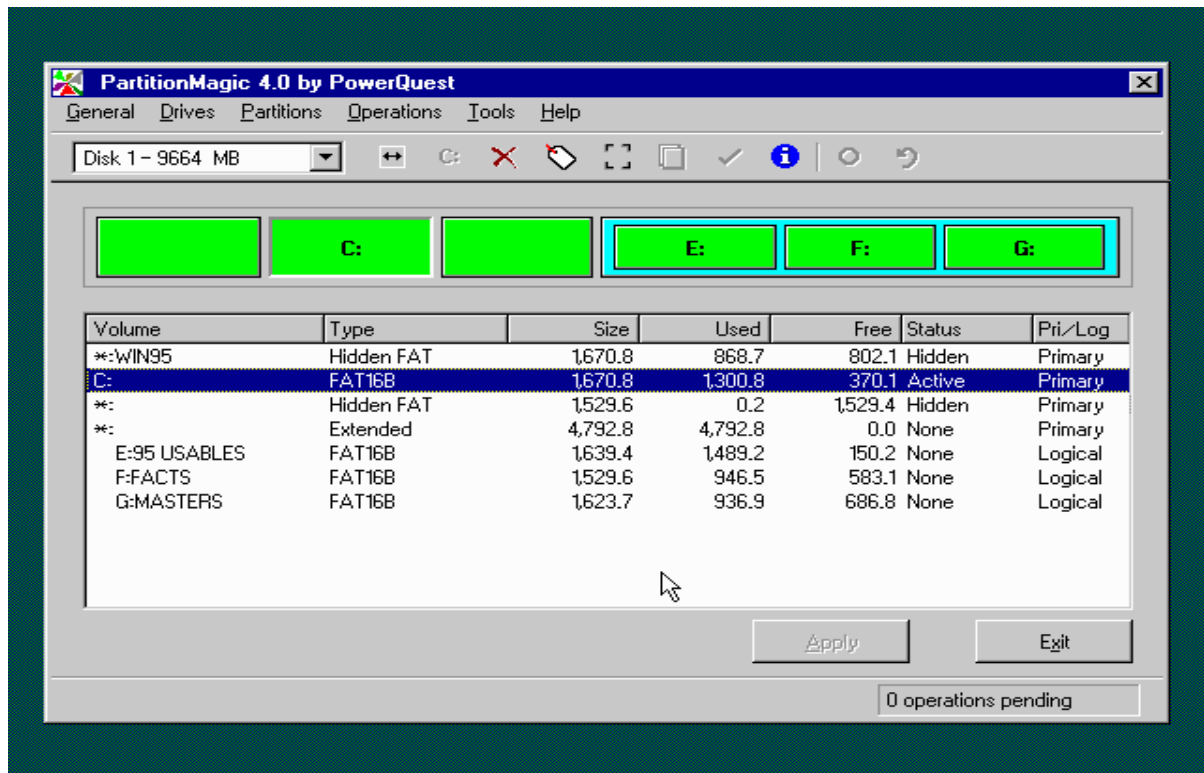


Figure 6.4.1: A screen shot of Partition Magic

Once the Live CD had booted a command line login prompt was available from which a root login was used. The following commands were used to mount a partition on the slave hard drive from where the perl script was run. The PC's name was "front" and each day a new text file was used to capture the output of the fingerprinting process.

```
[root@localhost ~]# mount /dev/sdb6 -t auto /mnt
[root@localhost ~]# cd /mnt
[root@localhost mnt]# ./fp.pl > front.01.txt
```

With this configuration it took approximately three weeks (twenty one days) to iterate over the primary master hard drive (/dev/sda in this configuration). In order to display this information gnuplot was used. In previous examples the graph had been plotted using a cross for each position (the default), but here many more sectors were processed per graph and, with this greater density of information, limbs of the crosses obscured other information. To overcome this loss of clarity the default cross was replaced with a dot this helped to keep the number of images down

while still displaying the overall information.

For the following set of graphs, the following gnuplot commands were used:

```
Terminal type set to 'wxt'
gnuplot> set terminal png size 800, 270
Terminal type set to 'png'
Options are 'nocrop medium size 800,270 '
gnuplot> set output "all8d.1.png"
gnuplot> set xtics 0,2475900
gnuplot> plot "all8.1.txt" with dots
```

These commands set the output to a PNG (Portable Network Graphics) format, 800 by 320 pixels starting at 0 and going 2475900 sectors, an eighth of the drive (this was modified for each subsequent image to give an even sized images) plotted with dots. These images have subsequently been edited to show the approximate partition boundaries. As with previous fingerprint images the X co-ordinates are the offset in sectors and the Y co-ordinates are the sum of that sector.

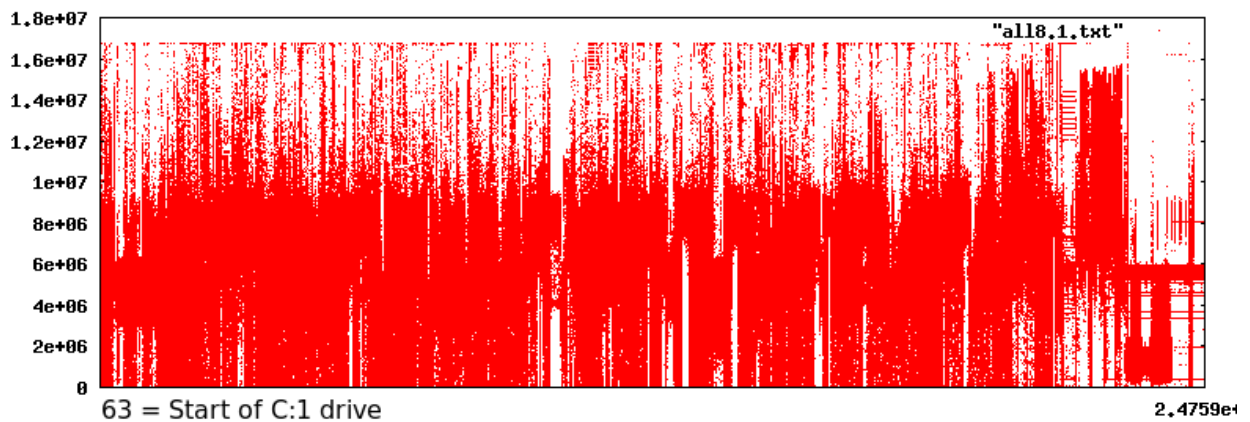


Figure 6.4.2: First eighth of drive

Figure 6.4.2 The first eighth of the drive shows the start of the previously used, but now broken, Windows 95B partition. The partition contains a mixture of executables and data. According to Partition Magic this disk was 52% used and 48% free. Toward the end of the partition, as illustrated in Figure 6.4.3, there appears to be unused disk space. Unused disk space in this drive appears to be a pattern on the disk that appears to contain about half ones and half zeros, and it is likely that this was the default pattern of this empty disk (ATA Model:IBM-DTTA-351010 Rev:

T560). This line can be seen at any point on the drive where there is little or no data. It is most obvious on the unused C:3 drive.

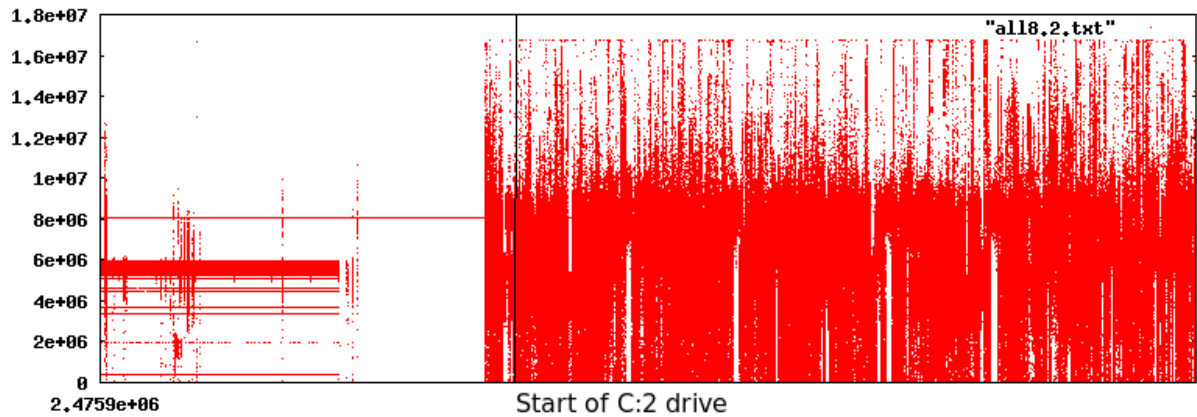


Figure 6.4.3: Second eighth of drive

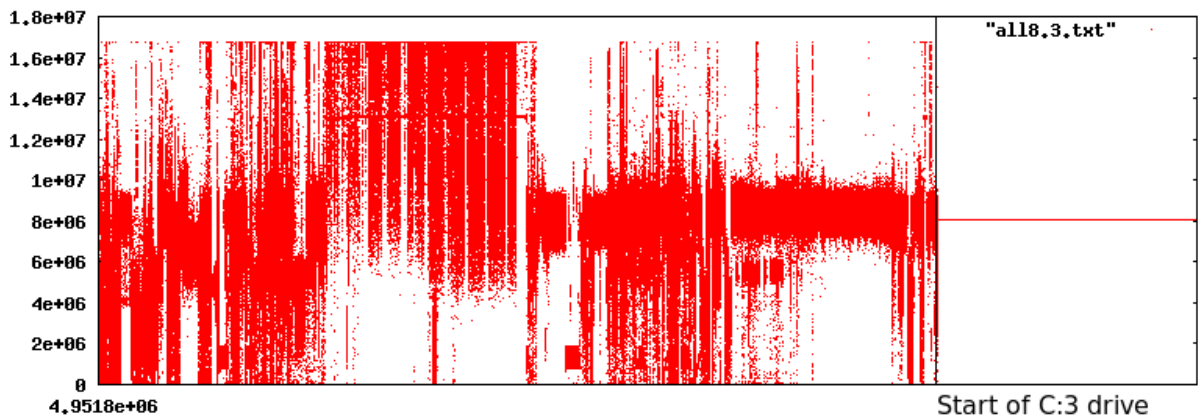


Figure 6.4.4: Third eighth of drive

Figure 6.4.3 and Figure 6.4.4 show the second C drive. This is similar in broad general pattern to the first drive; here, however, 78% is used and 22% is free. It is not known when this partition was last defragmented and although there are similarities between the drives there is no one to one matching of spikes or troughs. A closer comparison follows after the whole disk is considered.

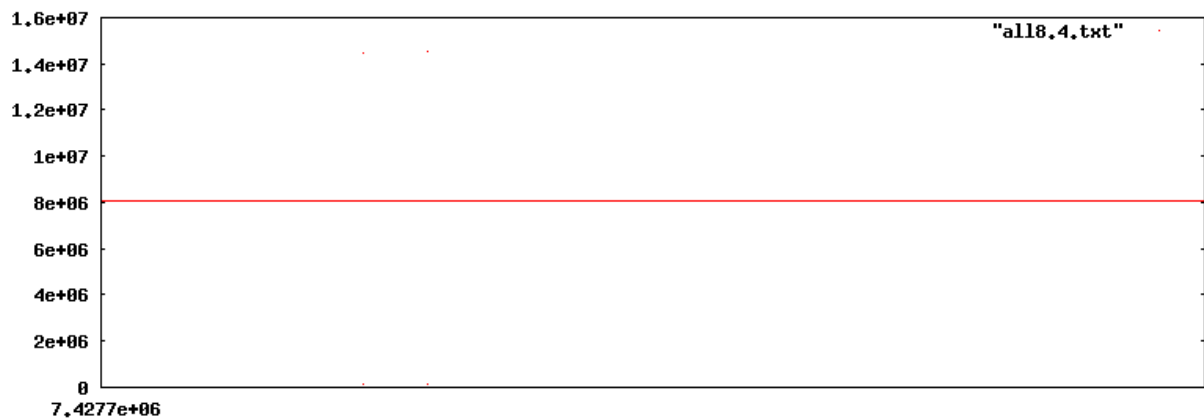


Figure 6.4.5: Forth eighth of drive

The end of Figure 6.4.4, all of Figure 6.4.5 and the start of Figure 6.4.6 show the third unused C: drive.

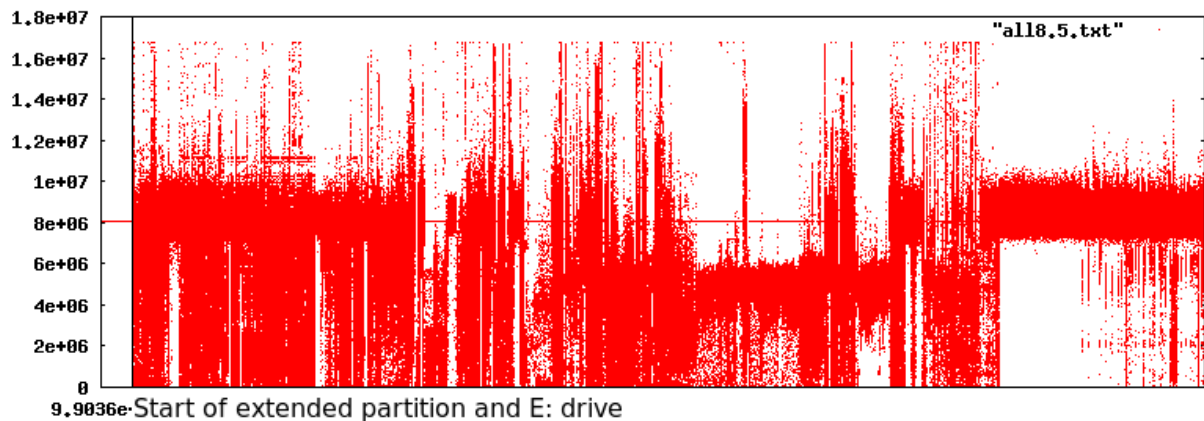


Figure 6.4.6: Fifth eighth of drive

The extended partition and the E: drive (starting 63 sectors into the extended partition) is shown in Figure 6.4.7. This and Figure 6.4.8 show that the E: drive appears to be mostly full, this is supported by the drive being 91% used. This drive was originally intended to store small useful programs, particularly those that did not require installation to be used by the current active C: drive, for example executables that could simply run from the directory. The exception to this was the installation of Microsoft's Visual Studio. Drive F: was intended for information and contains mostly textual data in the form of saved html pages and images. It is only 68% used.

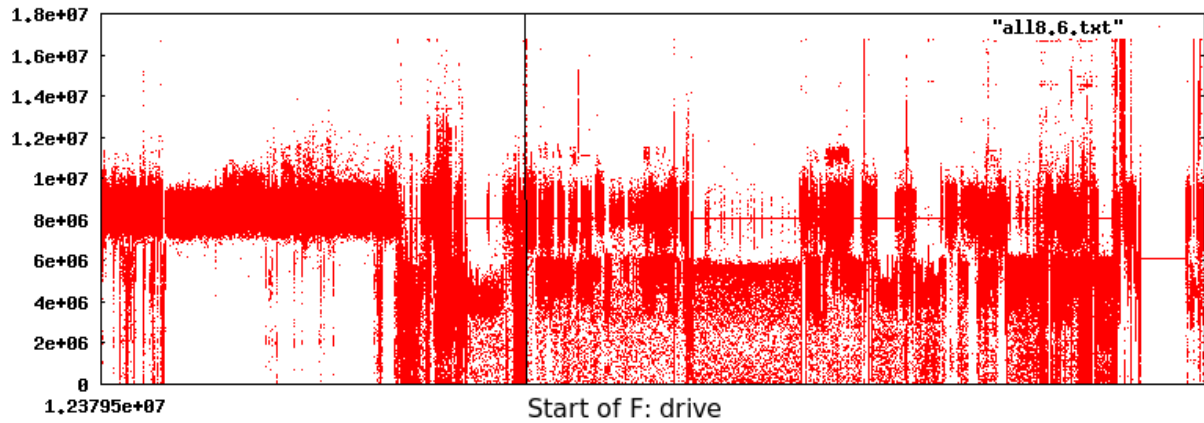


Figure 6.4.7: Sixth eighth of drive

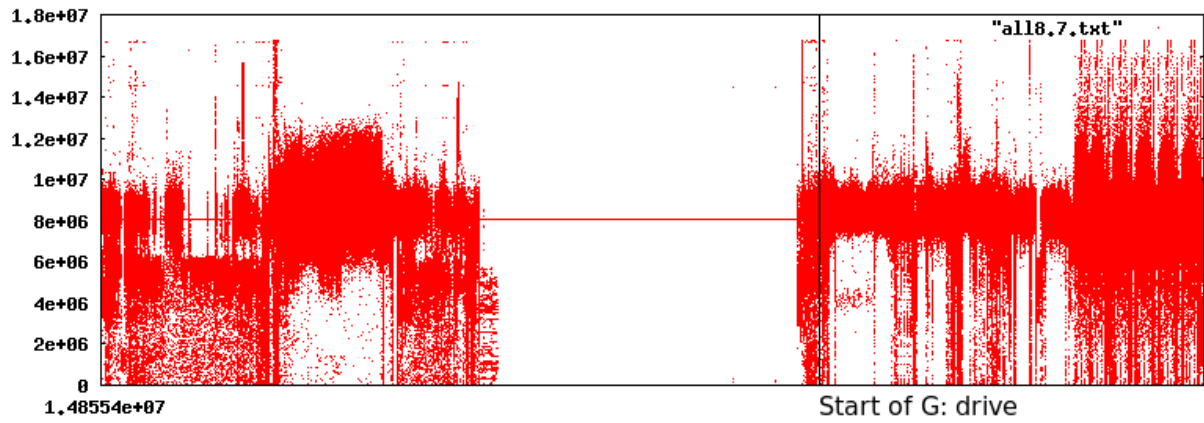


Figure 6.4.8: Seventh eighth of drive

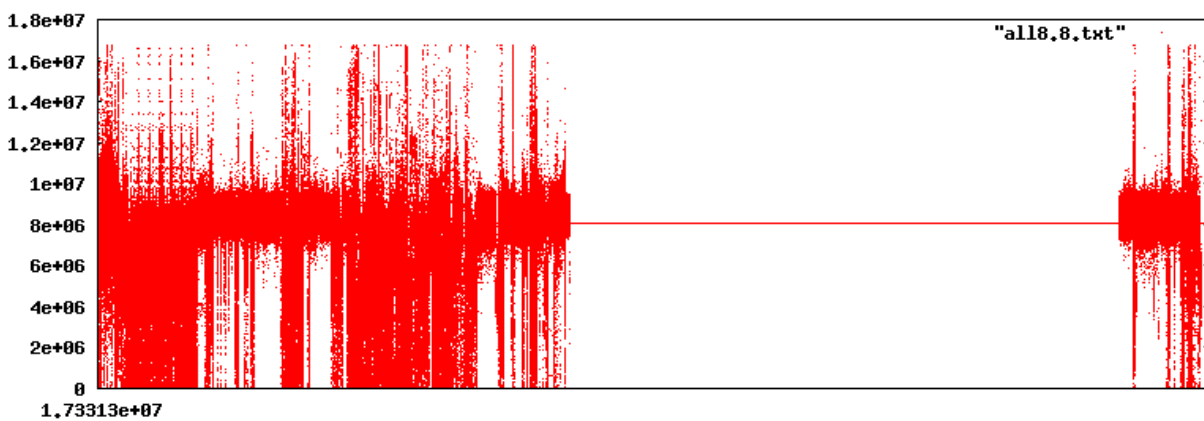


Figure 6.4.9: Final eighth of drive

The final logical drive, drive G:, contains master installation files. When the PC was installed the

CD used to install the operating system was first copied to the hard drive and the installation proceeded from the hard drive. One advantage of such an installation is that the machine will never prompt for the installation CD. Other installation files on the drive include modem driver files, Microsoft Office installation disks and service packs. The logic behind this approach was that the machine could be totally reinstalled from this partition and that all installation media could be kept in a separate safe location. It should be remembered that Microsoft Windows 95B pre-dates common availability of fast broadband and finding and downloading drivers and patches used to be more difficult. Drive G: is 58% used and once again there is a visible unused portion. It is not known why there is data at the end of this (or any other) drive.

If the regular pattern seen in the empty space is considered as “the line” then textual data appears to mostly fit below this line while executables spread from zero to maximum. This is however a very broad generalisation.

The following is the content of a sector of the blank portion of the disk that produces the line, this is the same for all sectors of the line. This was obtained by using `dd` to output the contents of one sector to `hexdump` using the `-Cv` arguments to format the output in a Canonical hex+ASCII display and to display all lines. This is presumed to be the normal empty state for this type and make of hard drive, the state in which it left the factory.

```

00000000  08 8c 03 9a 5f 78 76 94 8f 45 bf 49 e3 96 00 c0 |...._xv..E.I...|
00000010  88 9d dd c0 6d 36 60 df 48 5d ad f7 46 d1 32 24 |....m6`.H].F.2$|
00000020  38 29 95 cd ad 28 d2 a2 dc 89 f3 57 d9 21 cf de |8)...(. ....W.!...|
00000030  df 8e 1f d3 30 3e 86 19 64 1e 9c 2f 95 b4 d8 36 |....0>...d../...6|
00000040  55 df 4b 4d cd 24 fb 31 e8 b8 e3 bf 0d ff 6d 43 |U.KM.$1.....mC|
00000050  32 02 64 43 b4 69 12 a7 f4 53 cc da 02 eb 60 f7 |2.dC.i...S....`.|
00000060  63 dd d0 68 1b 97 b8 2a 3b 7c c9 14 04 ef d7 44 |c..h...*i|.....D|
00000070  cd 62 b3 34 da 54 73 82 3f 5d 58 dd 14 2c ec 47 |.b.4.Ts.?]X...G|
00000080  ef c1 8f 10 00 8d 18 a0 e5 ed a5 98 43 9a 4b 5a |.....C.KZ|
00000090  7d 7b e1 6c 8a 84 5b cf 1a 03 26 cb 54 17 de 3a |}{.l..[...&.T.:|
000000a0  09 8c c1 e4 ea 00 76 c9 7a 86 40 2e 59 96 56 28 |.....v.z.@.Y.V(|
000000b0  9c 84 86 5a b7 62 c8 db e2 7a e5 dc 56 31 e5 06 |...Z.b...z..Vl..|
000000c0  53 ab 63 21 48 cf 70 04 18 2d 2c 64 df 5a c9 7a |S.c!H.p... ,d.Z.z|
000000d0  09 1c 07 14 58 44 fb 19 73 57 04 f4 b6 ea f8 08 |....XD..sW.....|
000000e0  f1 f0 3e ba 9e c4 ef df 69 30 c2 6f 70 49 bc 3f |..>.....i0.opI.?.|
000000f0  2f 4c 90 5d 71 67 7f 28 3b 9d c7 9a 14 94 53 ca |/L.]qg.(;.....S.|
00000100  8a 95 e1 3c 6d 8c 1f 03 97 43 20 2e b8 af 91 95 |...<m....C .....|
00000110  fd 81 10 9b 09 ee 2a cd 2d b5 2c 01 1f 6f 7e f6 |.....*.-.,.o~.|
00000120  57 3f a0 e8 44 c3 7d 50 5a 84 31 25 63 bb f5 bd |W?.D.}PZ.l%c...|
00000130  ea 93 46 dd 34 e4 18 f3 c3 6f 04 03 8a a2 45 63 |..F.4....o....Ec|
00000140  13 fa 9c 9e b9 e7 c6 c4 f2 77 a7 83 2a 87 d6 21 |.....w...*...!|

```

```

00000150  f7 c2 b3 da 0a 3d ae ad  fe 05 e5 22 0d 3a bf 12  | .....=....." :. . |
00000160  03 34 bb eb 67 5e 01 85  21 09 f9 1c ce 17 6f 55  | .4..g^...!.....oU |
00000170  ab b2 a1 f4 ac d9 95 38  67 18 26 86 70 2d 49 29  | .....8g.&.p-I) |
00000180  f4 cf ad 00 f7 7c 80 e7  3a 90 65 6f 11 56 43 12  | .....|...:eo.VC. |
00000190  19 75 22 2e 43 77 c2 fd  13 b3 f3 00 7c 5a 89 f8  | .u".Cw.....|Z.. |
000001a0  36 07 e7 be 11 77 db 5f  11 c9 fb a1 c9 13 1a 3d  | 6....w._.....= |
000001b0  da 81 14 3d 00 c7 70 83  9d 42 33 0c 02 87 6d f2  | ...=.p..B3...m. |
000001c0  aa 8e a9 a4 70 6f ef 91  04 d8 84 7f c9 0a 08 df  | ....po..... |
000001d0  09 b5 18 76 1e 59 23 82  24 a4 9e cc e3 61 2a b6  | ...v.Y#.$....a*. |
000001e0  b2 72 fa e0 ce 6c e6 43  fd 4a 9f 7f f1 de 66 28  | .r...l.C.J....f( |
000001f0  54 58 99 d9 df ad aa d6  55 14 b0 04 fe 7f 40 41  | TX.....U.....@A |

```

6.5 Comparing Two C: Drives

With whole of the hard drive iterated, a more in depth comparison was attempted of the two windows installations. The two installations of Microsoft's Windows 95B were plotted as discreet graphs and to do this the xtics range (the X axis range) was set in gnuplot to reflect the start and end of each logical drive. The two drives plotted in their entirety are shown in Figure 6.5.1: C:1 and Figure 6.5.2: C:2 respectively.

When seen side by side there are both obvious differences and almost similarities. Both C: drives appear to have data at the very end, but this may not be an active part of the file system and may have been generated during the installation or any other common process. The first halves of the installations appears to be broadly similar. Although most of the sectors seem to have a sum in the lower two thirds, there is some clustering at around the halfway point or “the line”. It is expected that this clustering may be due in part to the fact that any sector to which no data has been written will still contain the original pattern. The same can be said for the unused portion of sectors that were only partially written, thus the sum of these sectors will tend toward the empty state.

The first C: drive (C:1) has several straight lines for most of the final third. As these are believed to be beyond the current extent of the operating system it is possible that they may have been present on the drive initially or be the result of some operation that occurred in the past. The second C: drive (C:2) contains an area of high average values about two thirds from the start. This situation may be due to the fact that data that is currently there is represented with more ones

than zeros. Alternatively it may be a consequence of something overwriting the default empty state with a pattern containing a higher percentage of ones. Therefore the unused space would have a high sector sum. The presence of a barely visible second line in this portion of the graph tends to support this explanation, but there is insufficient information to confirm this.

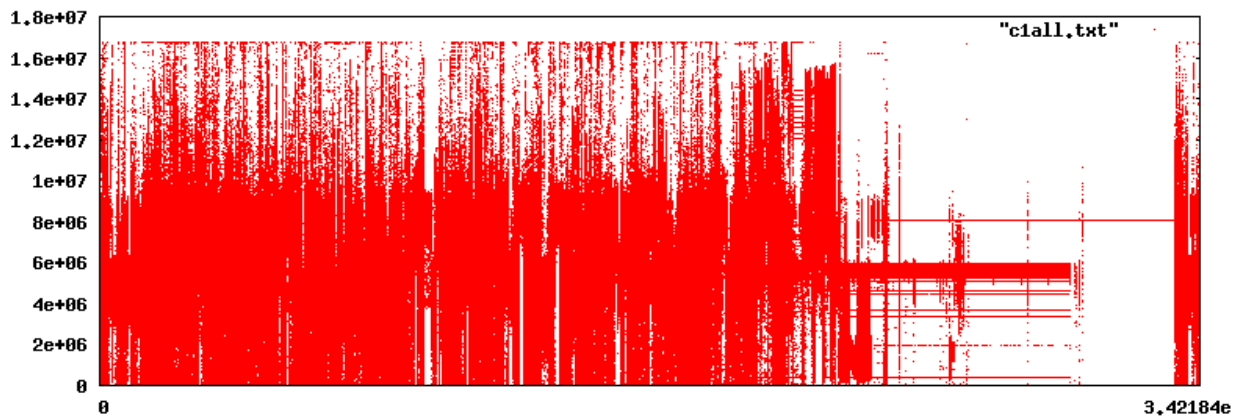


Figure 6.5.1: C:1

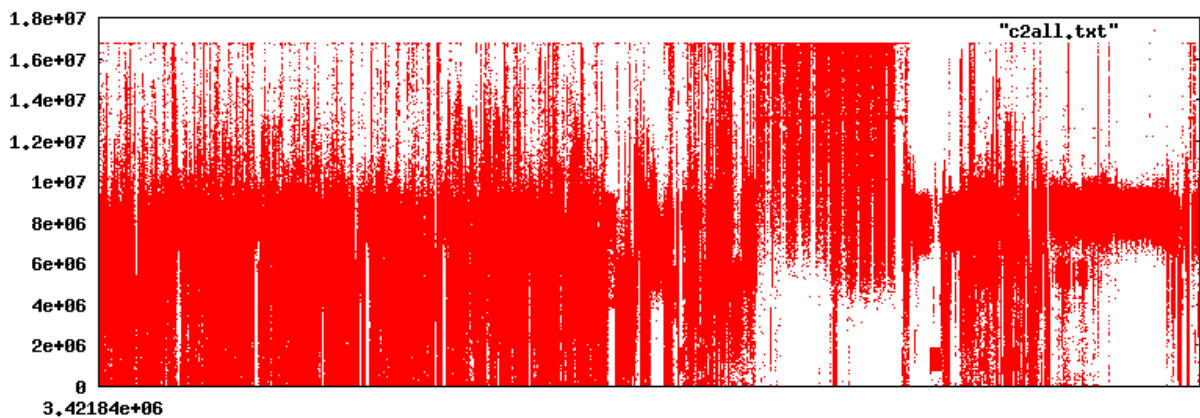


Figure 6.5.2: C:2

As the disk was not fingerprinted prior to any installation being performed, caution should be used when trying to draw any conclusions about the current state. It was thought that two identical installations would have originally put the same files at the same offsets, but if the drives had been defragmented then a small variation in file positions may have propagated producing much larger differences.

The initial 1000 sectors of each partition were plotted to see if the start of the logical drives were similar. As the density of the data was significantly reduced, gnuplot was left in the default state for plotting the data points using a cross. The following two figures (Figure 6.5.3 and Figure 6.5.4) show these first sectors for C:1 and C:2 respectively. Despite expectations of some similarity, it is clear that these are not the same. The first half of each of these images are the FAT (File Allocation Tables) and each drive had two identical copies (one is a backup). Just as the files were different so too were the FATs that referenced them. Some regions of similarity between the second halves of the image could suggest similar file(s) in similar positions, but this is not conclusive.

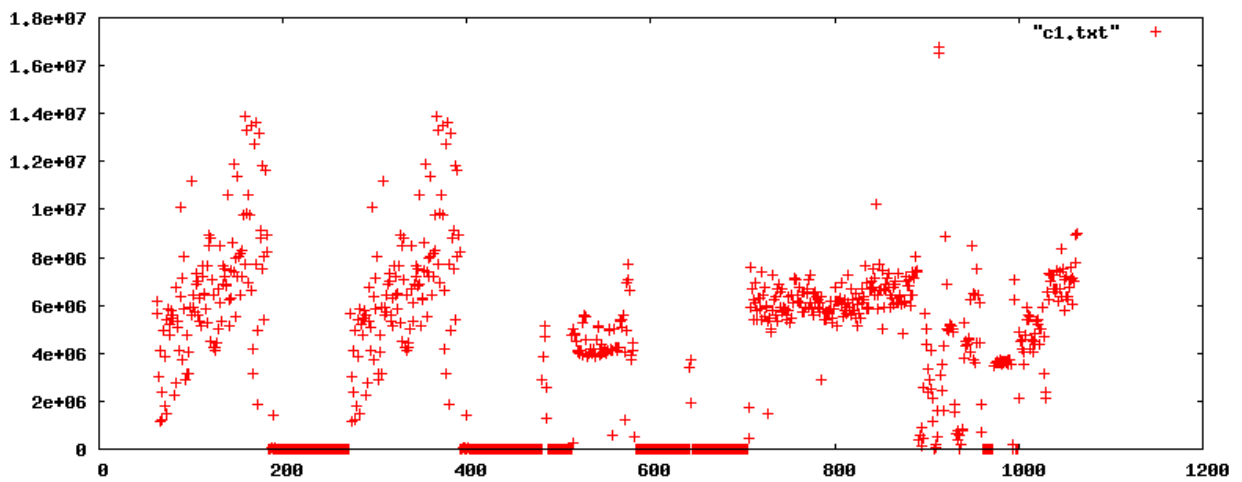


Figure 6.5.3: C:1 start

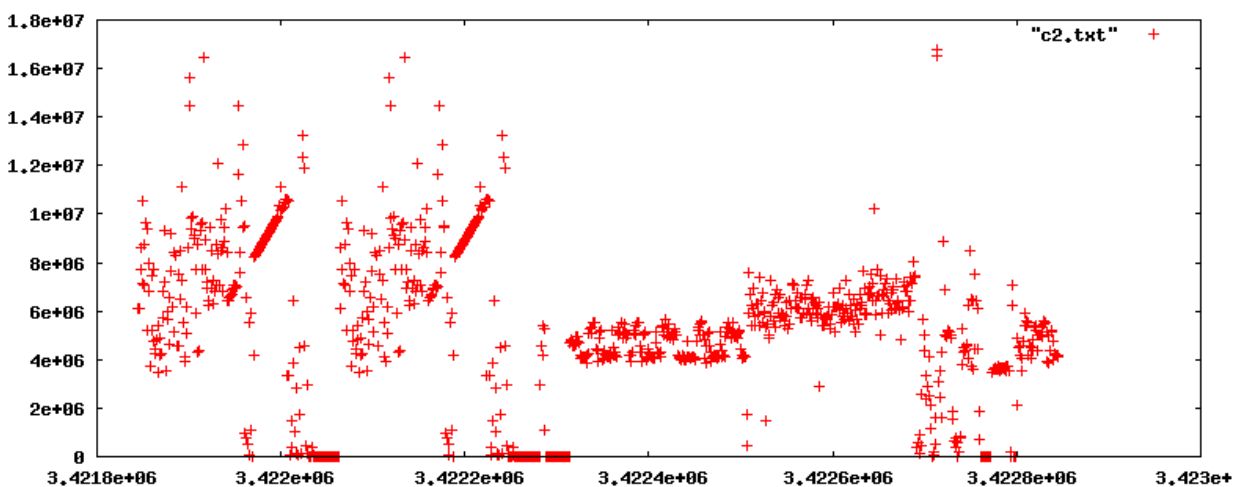


Figure 6.5.4: C:2 start

One of the biggest differences between the two drives' FATs, namely the length of the lines of zeros at the bottom of the images, can be explained by the differing utilisation: C:1 at 52% (which should leave 48% of this area as zeros) and C:2 at 78% (which should leave 22% of this area as zeros). Why there was another area of zeros roughly between sectors 590 and 700 on drive C:1 is not known. (It is possible that this was the reason why this installation would not boot to Windows 95B. A more probable explanation could be the fact that some manipulation was performed on the boot files shortly after installation to retain the ability to boot to a previous version of DOS, as was possible with the original version of Windows 95A.)

6.6 Formatting a FAT 16 Drive

In theory if the first partition was formatted then the FAT should contain all zeros. Given that it took three days to retrieve the information about the whole of the disk and a format takes only minutes, the rest of the disk may be unchanged.

The drive was set to be the active primary partition and was booted to the command prompt only (this is as far as it could boot as the normal Windows starting option resulted in a blue-screen). Unfortunately the chance to prevent the partition attempting to boot into Windows was missed and the PC hung before it was restarted with the command prompt only. Then it was formatted using `format.com` from the `Windows\Command\` directory using the command `"format C:"`. This file was 49,543 bytes in size and had a creation time of 24-08-96 11:11:10 (the version distributed with Windows 95B). After the formatting was finished it was realised that `format /s` would have been a better choice as in this configuration the PC could not boot from the hard drive so a copy of a DOS 6.22 installation floppy disk was found and used to `sys` the drive (the `sys` command copies the bare minimum of files needed to boot to the target device). Although it was not a requirement for this research that this partition was bootable, it would have greatly simplified any future work done with this drive. The Live CD was placed back in the CD drive, booted to Linux and the drive reprocessed. Although the addition of some system files from the old version of DOS was undesirable, the sizes of the files were minimal and would not have had significant impact overall. Their presence meant that caution should be taken before making any

assumptions about the start of the drive beyond the location of the FAT sectors. Their presence also meant that there must be at least a couple of entries present in the FAT tables.

The results from the post formatting fingerprint and the original were compared. A Unix `diff` showed that up to sector 1068 there was hardly any original data remaining and after that only 38 sectors were different, all of which were before sector 149250 (this drive occupied 3421844 sectors). Before the drive was formatted there had been a delay in pressing the F5 key to gain access to the command prompt and the PC had hung and needed a power reset before another attempt could be made to break into the boot options. It is suspected that some of the 38 later sectors may have been written to at this point but there is no evidence to support this.

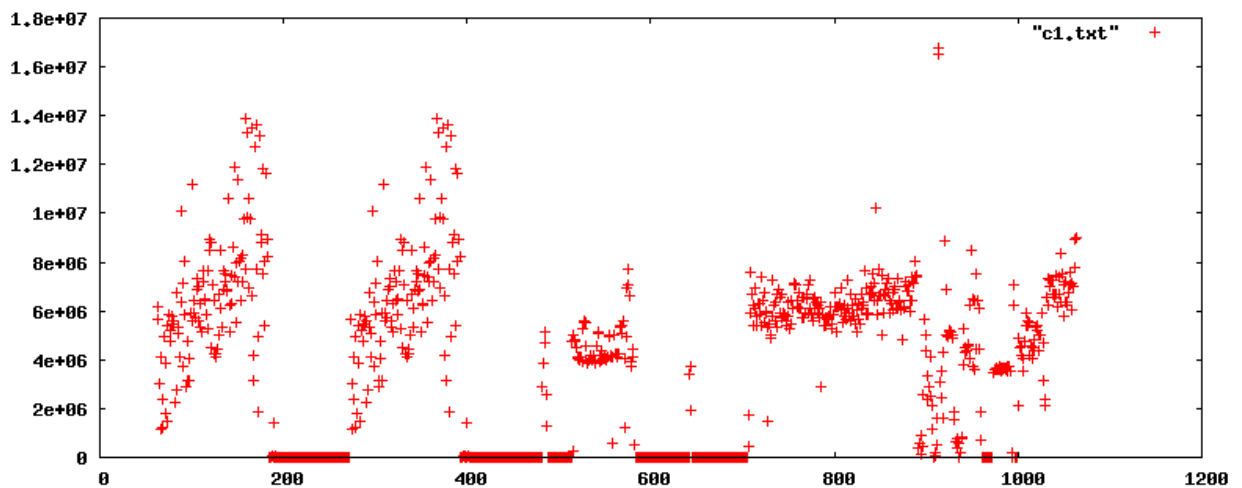


Figure 6.6.1: C:1 before format

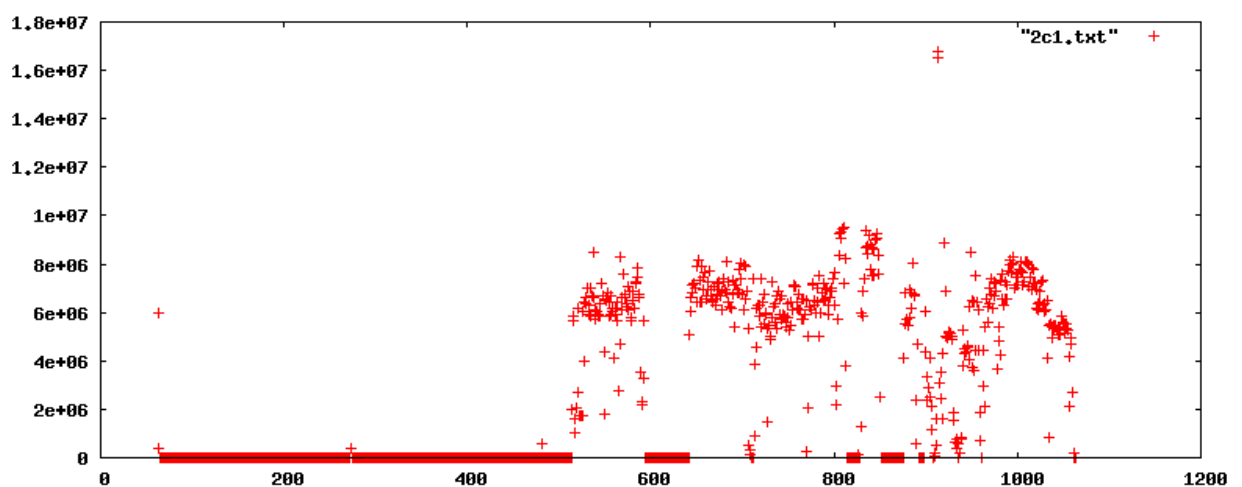


Figure 6.6.2: C:1 after format

The previous figures (Figure 6.6.1 and Figure 6.6.2), of the first 1000 sectors, show the before the format and after the format graphs indicating that the FATs have been zeroed out except for the system file entries. (Figure 6.6.1 is a repeat of Figure 6.5.3 before format, allowing side by side comparison.) This also shows differences in the subsequent sectors that are most likely due to the `sys` command placing different boot files in this location; these differences are to be expected.

The picture for the whole of this drive as shown in Figure 6.6.3, is a repeat of Figure 6.5.1, allowing a side by side comparison. Figure 6.6.4 is the post formatted drive.

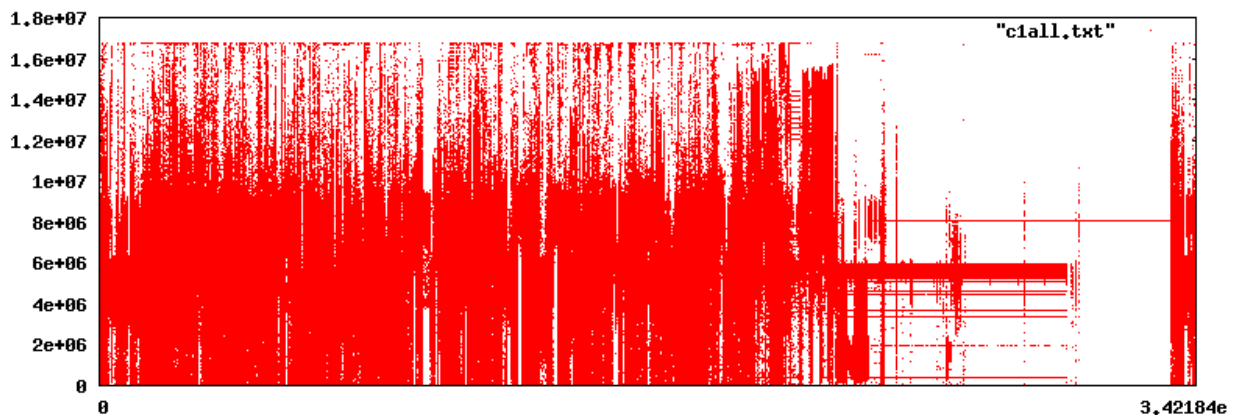


Figure 6.6.3: C:1 pre format

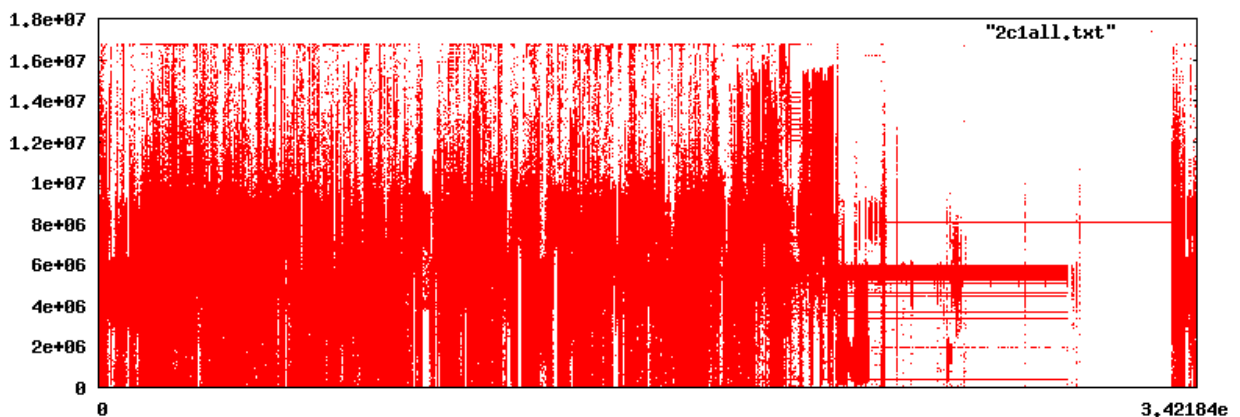


Figure 6.6.4: C:1 post format

It is commonly thought that if a hard drive is formatted all the data would be removed. The formatting described was done using the Windows 95B version of `format.com`. Newer versions

shipped with later Microsoft products may have different capabilities and offer different switches, but these have not been tested. Although the fingerprinting technique does not discover anything new, it does allow a good visualisation of what happens to a hard drive during the format procedure and of how little data is removed.

The drive has been left in this state as it may be interesting to “recover” the data at some future point. Although the examination of these drives was done using a live CD which should not change the contents in any way, both the slowness in preventing windows attempting to load and the usefulness in having a bootable system meant that although the basic concept of zeroing out the FAT tables has been shown to be fundamentally correct, extreme caution should be used trying to interpret any change on the disk outside these sectors.

6.7 A Linux Disk

While the events of the previous section were taking place, a similar process was being performed on a PC running Linux. This machine was running RedHat Fedora core 6 and had been in use for several years. The first operating system that had been installed was RedHat Linux 7.2, but over some years this had been upgraded and reinstalled several times. On this occasion the fingerprinting was performed from and recorded within the operating system which meant that the contents of the disks were changed by the recording process. The decision to work in this fashion was largely dictated by the fact the CD drive from this machine was being used on the old Windows PC and that the version of Fedora core used was too old to recognise the NTFS (New Technologies File System) formatted USB (Universal Serial Bus) storage device for saving the data off the machine. Although the process would write approximately 4GB of data to the file system there were about 95GB unused on a 146GB / (root) partition and it was not considered that the change to the data on the disk would be significant. The basis for this was that this investigation was simply aimed at observing what could be identified. If user data was found on the disk in areas where it were expected to be, then it was not significant as to whether it was written before or during the investigation. Additionally as this installation occupied two hard drives of 40GB and 122.9GB, it took 101 days to iterate in their entirety and there was

considerable convenience in storing the generated data on the machine itself. It could have been possible to export and mount a drive from another machine, but it was believed that this would slow down the process (not tested) and also no significant storage space existed on another machine.

The drives had been in use for some considerable time and it was realised that the information retrieved would show not only the currently available files but also files that had been deleted or updated. (The following section 6.7 considers a fresh install on a disk that had zeros written to all sectors)

An `fdisk -l` of the first drive gives the following information:

```
Disk /dev/hda: 40.0 GB, 40020664320 bytes
255 heads, 63 sectors/track, 4865 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

Device	Boot	Start	End	Blocks	Id	System
/dev/hda1	*	1	13	104391	83	Linux
/dev/hda2		14	4865	38973690	8e	Linux LVM

Here we can see that the first partition, the bootable one, is 104391 blocks, occupies 208782 sectors and is of type ext3. This partition contained the files needed to boot the PC to fc6 (Fedora core 6) while the second partition was controlled by a Linux LVM (Logical Volume Management). This volume spanned both hard drives and managed both the swap and / file systems irrespective of the number of physical drives present.

The first data to be processed from this drive was the boot partition. As before this data was plotted using gnuplot. The commands used here were:

```
gnuplot> set terminal png size 800, 270
Terminal type set to 'png'
Options are 'nocrop medium size 800,270 '
gnuplot> set output "box-boot.png"
gnuplot> set xtics 0,209000
gnuplot> plot "box-boot.txt" with dots
```

The resultant graphic is shown in Figure 6.7.1.

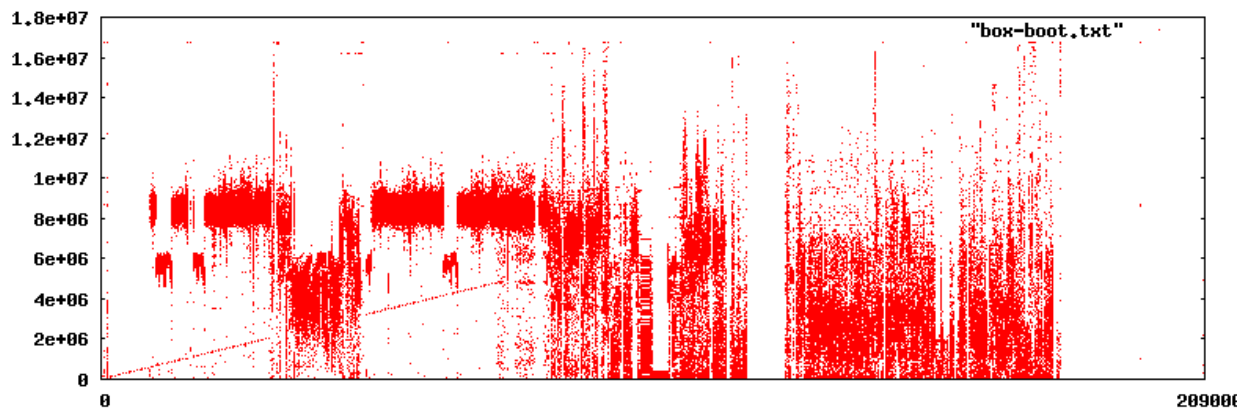


Figure 6.7.1: Boot partition

Although it was interesting to see the diagonal line starting at the bottom left corner, proceeding for just under half the partition and rising to about a third of the sector occupancy, a lot of caution is needed in interpreting data as it is not known what proportion of the data was active data and what proportion was overwritten or deleted data.

The whole of the drive was fingerprinted and like the boot partition, many areas had the diagonal stripes. Other patterns as can be seen in the following image Figure 6.7.2.

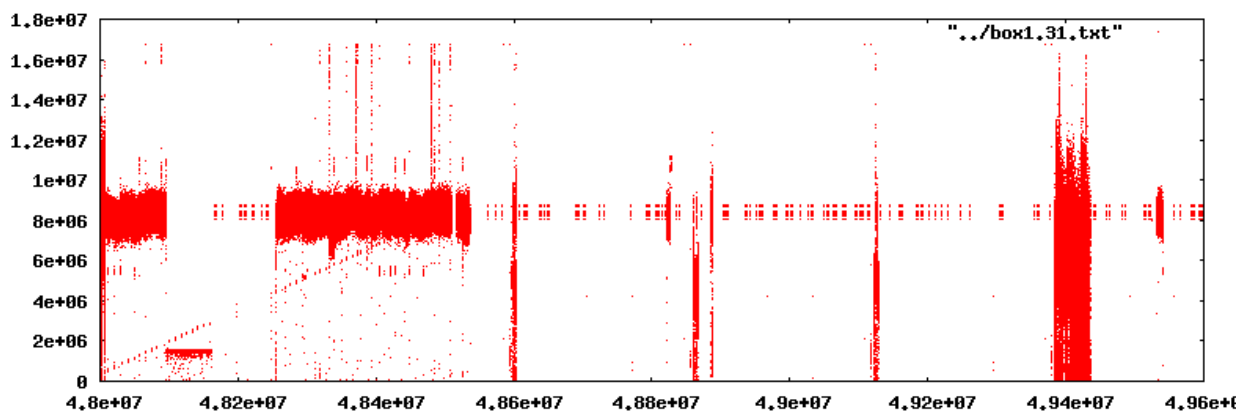


Figure 6.7.2: Some typical patterns

Typical patterns include these diagonal lines and a fence-like structure. The upright “posts” of the fence could last approximately 2,000,000 sectors at a time. Despite having collected a vast amount of data, very little information could be interpreted. An exploration of parts of the disk where a “post” was located indicated that this portion of disk was occupied by .tcl (Tool

Command Language) files. Although the contents were readable and the presumed file name was included in the comments, it was very quickly realised that the vintage of this data could not be established and no files in the current file system matched the disk information. As with previous investigations the data on the disk was not the same as the data in the current file system.

6.8 Two Fresh Installs of Linux

Although historic data was available, and potentially being useful for data recovery, this data was not permitting clear snapshots of the current Linux system. Some hexdumps were taken from differing patterns of the drive and saved for possible later examination. The second (larger) hard drive was left untouched, but the whole of the first hard drive was zeroed out using the following command:

```
dd if=/dev/zero of=/dev/sda bs=512
```

This command wrote zeros to the whole of the disk and provided a blank disk in a known state on which the next series of experiments could be performed. The PC was then booted from a Partition Magic v8 CD and all but the first 5GB (4981.1MB) of the drive was partitioned as a primary NTFS partition. NTFS was chosen as it is able to be read by other operating systems (if required) and was able to easily handle a partition size of 33181.1 MB. (FAT16 cannot handle above 2G and although FAT 32 can be used above this size there are inefficiencies in the linked list structure as the amount of data grows. Even with FAT32 there is a 4G file size limitation which, at the time the partition was created, was not known if this would be a problem).

The PC was then booted from a Fedora 9 i386 DVD and a default installation was performed. The only exceptions in this case were that the option for installing office and productivity tools was unchecked (not installed) and UK keyboard and local were selected (from their default US values).

Once installed the PC was logged into as the user created during the install and, from a console, “su” was used to gain root privileges to mount the NTFS partition. The NTFS partition was

mounted to `/mnt` using the `mount` command. After moving to the NTFS partition the fingerprint script was run creating files on that partition.

```
[root@localhost sue]# mount /dev/sda1 /mnt
[root@localhost sue]# cd /mnt
[root@localhost sue]# ./fp.pl > lbox.f9.01.txt
```

As with the previous scan of the disk containing Linux, the operation was performed using the operating system under examination. On this occasion the output data was written to a separate partition. Due to logistical reasons the PC was powered down each evening and separate files were created each day. Although the data was being written to a different partition there would still be changes to the Linux partition as each boot would modify the last accessed time on some files and system processes would generate log files, however since each sector was examined as a snapshot of its current state, the fact that there was a difference of five days between the first sector being examined and the last was not considered significant. During the installation the option to encrypt the disk was observed, but not chosen; after the plain installation of Fedora 9 and the fingerprinting process was completed, the start of the disk was once again zeroed out this time using the command:

```
dd if=/dev/zero of=/dev/sda bs=512 count=10201120
```

This command which wrote zeros to the disk, stopped just short of the NTFS partition. Unfortunately it also overwrote the partition table and removed the NTFS entry. The entry was then hand edited back, but on the wrong line hence although the NTFS partition was the same as before, its reference in the partition table was that of the second entry. After the subsequent installation, it would be `/dev/sda2` (not `/dev/sda1` as before). This error was not noticed until after the installation, however it is not considered to have had any impact on the experiment.

The installation of the Fedora 9 was repeated as before. This time however the encryption option was selected before the drive was once again fingerprinted. The following images show the start of the two installations side by side with Figure 6.8.1 and Figure 6.8.2, (the boot partition is approximately the first quarter of the image). The boot partition for the Linux installation can not be encrypted so there are some close similarities between these partitions, although a portion of

the data does appear to be located nearer the start of the disk on the encrypted partition.

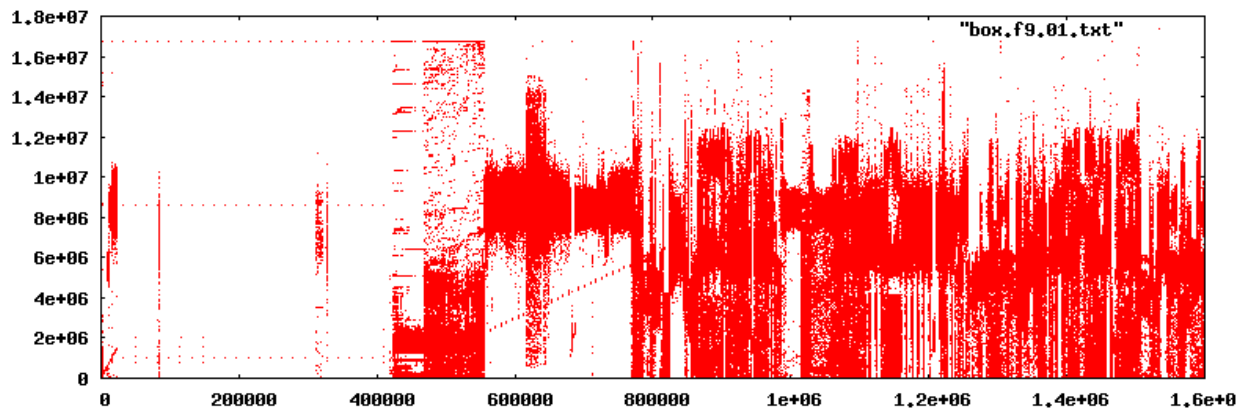


Figure 6.8.1: Boot partition and start of LVM unencrypted

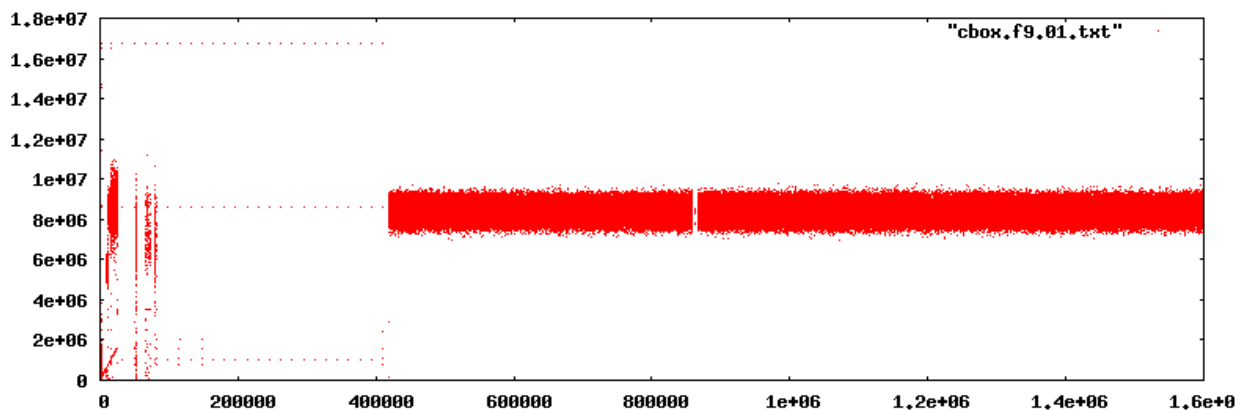


Figure 6.8.2: Boot partition and start of encrypted LVM

Both the boot partitions are of type 0x83, Linux ext3. One possible reason for the differences between the two boot partitions was the difference in the partition table entry for the NTFS volume, however a subsequent zeroing out and reinstallation of another unencrypted installation (with the NTFS volume still as the second partition) discounted this possibility. Although the differences between the two boot partitions are interesting, the most obvious differences are in the main body of the LVM partition. Here the encrypted partition forms a narrow band of data while the unencrypted displays a more varied output.

The remaining parts of the unencrypted LVM partition are shown in Figure 6.8.3 to Figure 6.8.8. This is shown in its entirety as the installation is small enough to make this feasible.

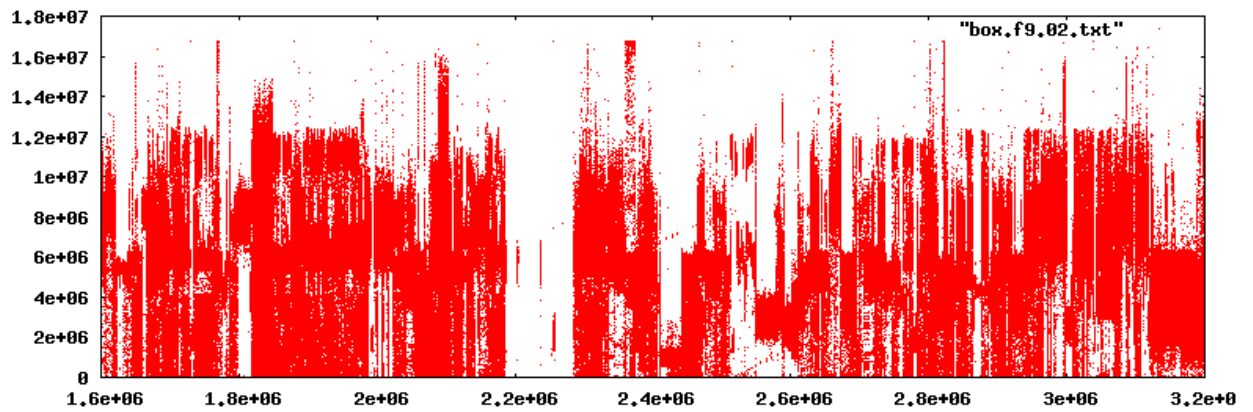


Figure 6.8.3: Unencrypted LVM part 2

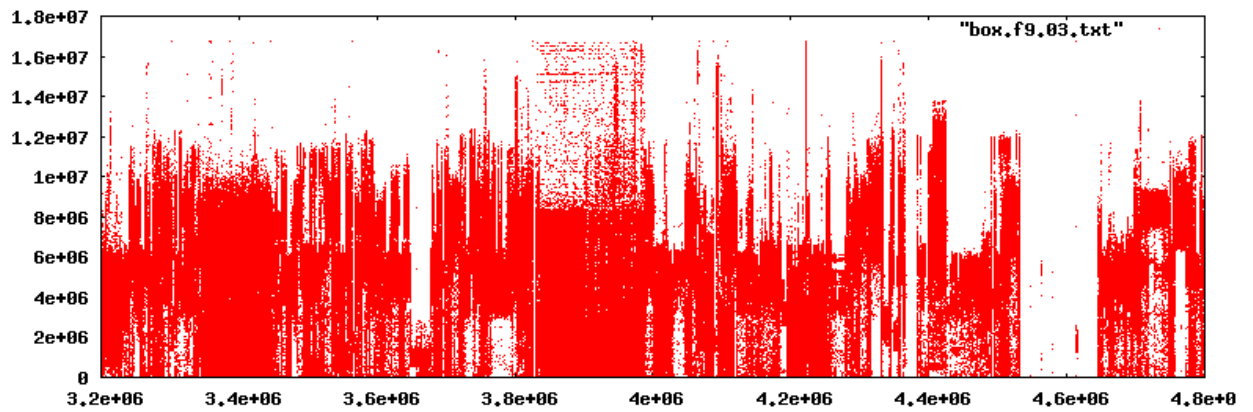


Figure 6.8.4: Unencrypted LVM part 3

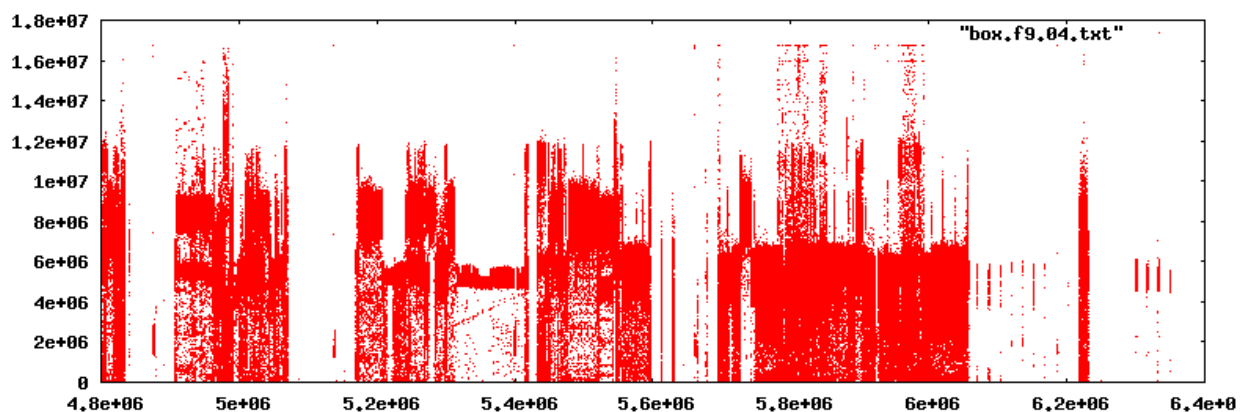


Figure 6.8.5: Unencrypted LVM part 4

The mixture of executable, text and graphics are not dissimilar to that of the FAT 16 Windows

95B installations. However the installation of the base operating system (of Linux) is more evenly distributed throughout the disk.

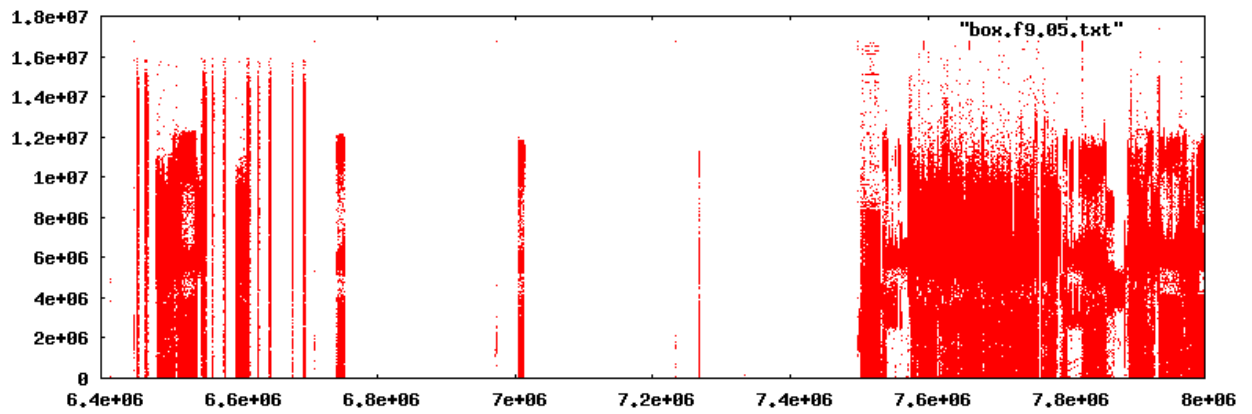


Figure 6.8.6: Unencrypted LVM part 5

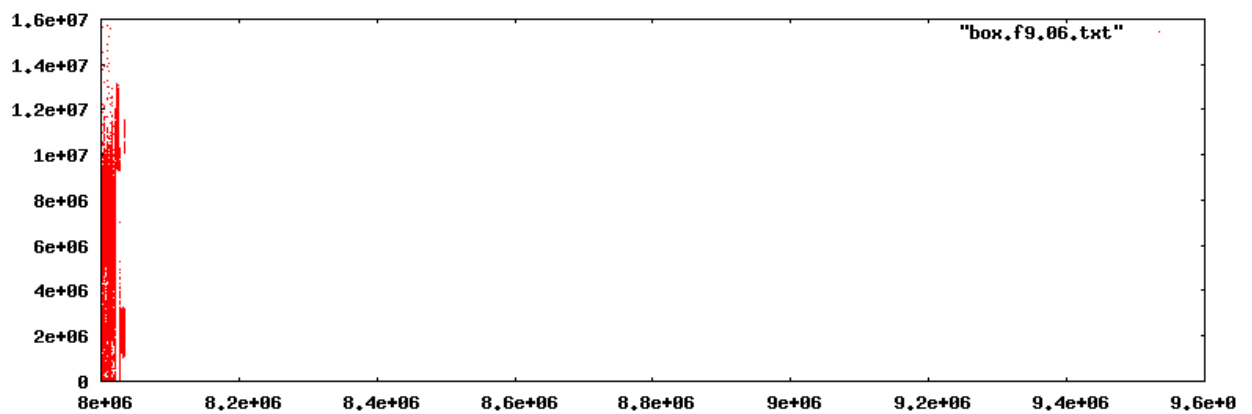


Figure 6.8.7: Unencrypted LVM part 6

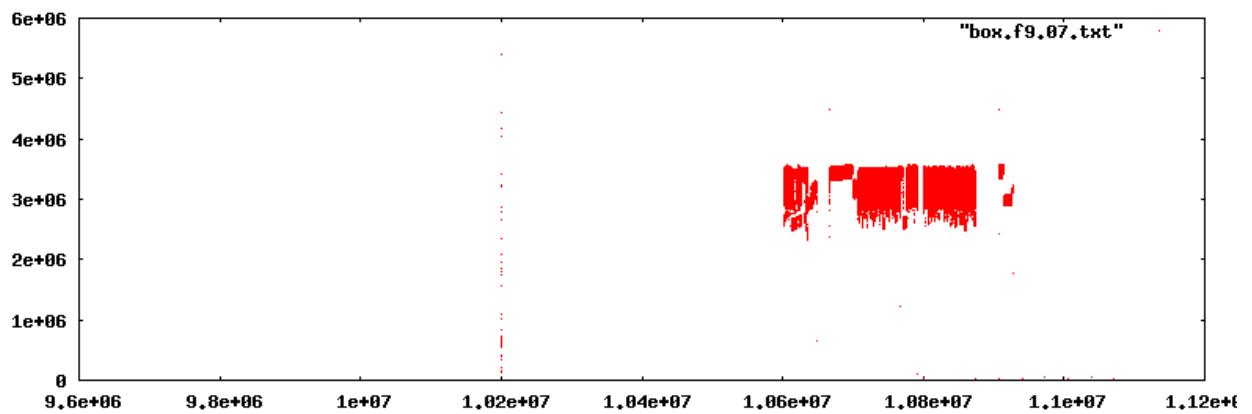


Figure 6.8.8: Unencrypted LVM part 7

The last of the data appears at the start of Figure 6.8.7. At the end of the LVM partition can be seen a little before halfway in Figure 6.8.8. The final part of this image contains the start of the NTFS partition and the data visible represents the data files of the scan. As with the previous Linux fingerprint it is possible to see some diagonal lines; while the fence like structures are not so visible, the “posts” or islands of data are very clear.

The following images are a repeat of the last installation, but showing the encrypted LVM partition. Caution should be taken if trying to compare these directly with the previous unencrypted ones as gnuplot had adjusted the scale as the maximum values are considerably smaller than that of the unencrypted installation.

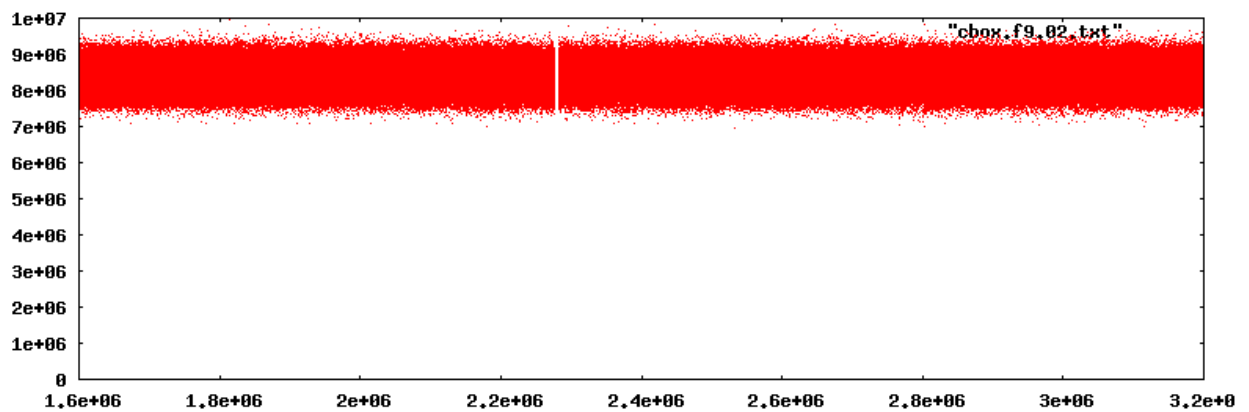


Figure 6.8.9: Encrypted LVM part 2

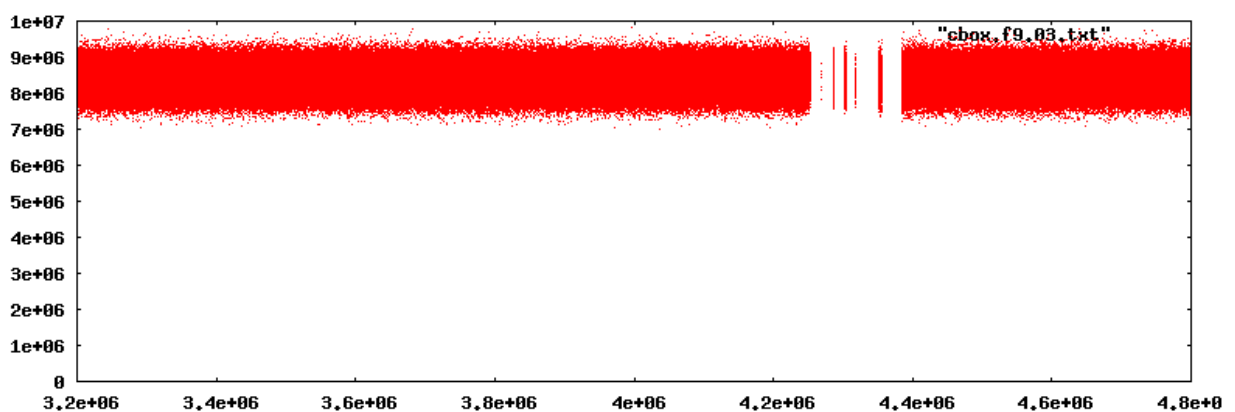


Figure 6.8.10: Encrypted LVM part 3

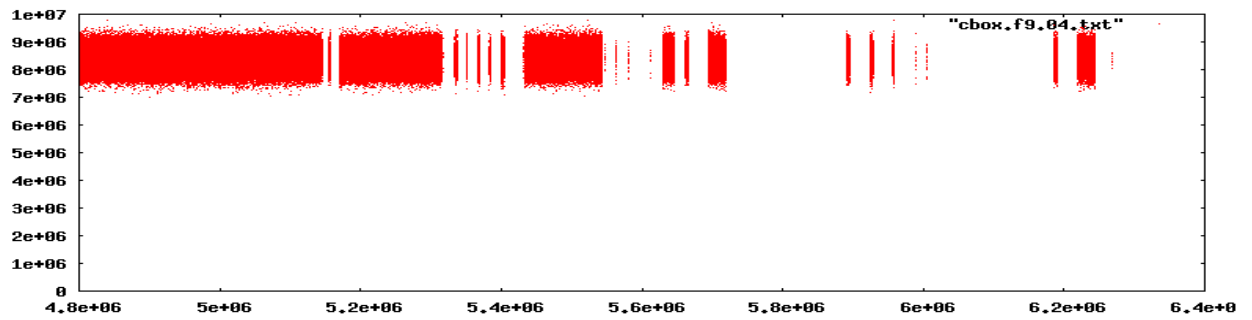


Figure 6.8.11: Encrypted LVM part 4

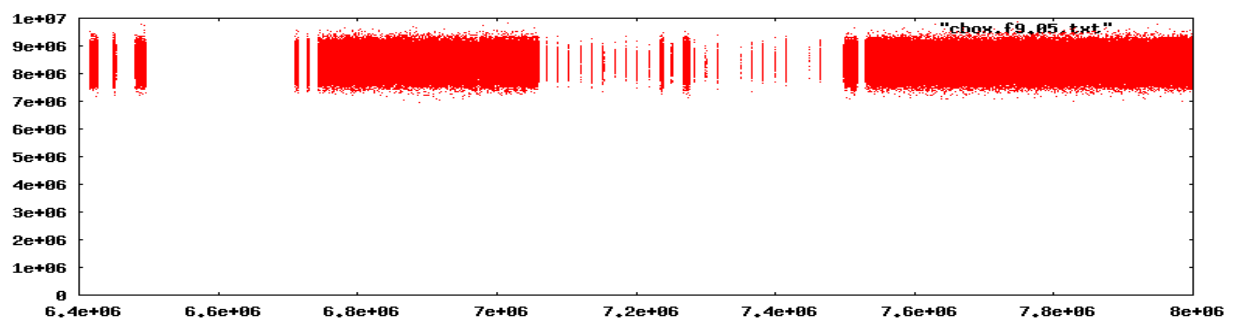


Figure 6.8.12: Encrypted LVM part 5

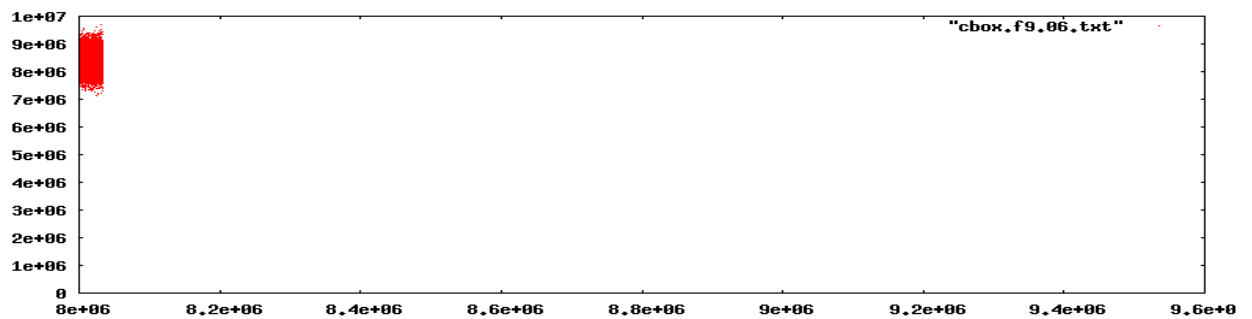


Figure 6.8.13: Encrypted LVM part 6

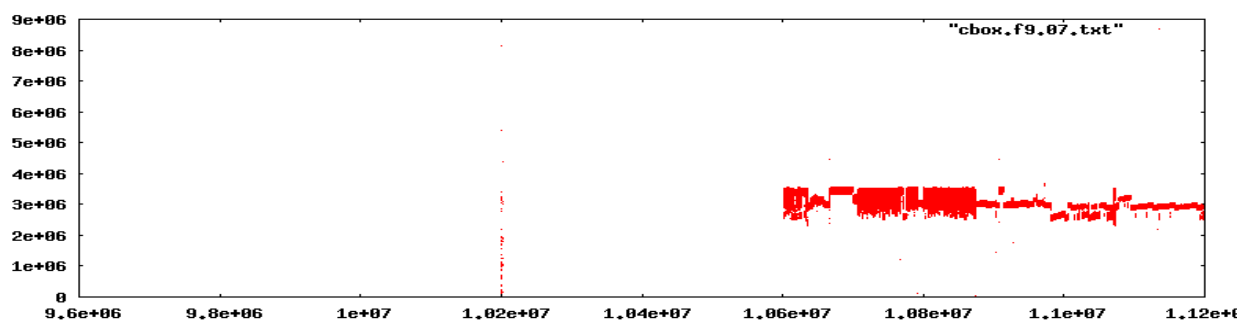


Figure 6.8.14: Encrypted LVM part 7

These images are interesting not for what they show, but for what they do not show. The only information that can be determined is if there is data stored on a portion of the file system or not. The overhead of such encryption is the prompt for a password before the PC will finish booting. While it is assumed that there must be some additional overhead in encrypting and decrypting data to and from the disk, it was not noticeable while using the machine. In the final image (Figure 6.8.14) it can be seen that more data was present on the NTFS partition. As the NTFS index has grown, the scale of this image is not the same as its unencrypted counterpart.

While the encryption was expected to obscure the data stored on disk, it was anticipated that there would be some visible difference between different types of files, but the minimal difference between sector sums was not expected. As with the previous scanning of these partitions the operation was performed from within the installed operating system and there was a risk that the operating system was modifying the reading of the disk to produce a degree of obfuscation. It was not believed that this was the case, but the PC was booted to a Live CD and the first sectors scanned again. The results from this scan were not a perfect match to the previous one, but this was to be expected as the encrypted installation had been booted a few times and there would be differences in the file last accessed attributes and log files would have been generated. Despite these differences the same overall pattern was observed, i.e. that of a narrow band of variation where data was present and all zeros where no data existed.

Encrypting disks had not previously been considered, but looking at these results the extra security provided means that this option has some very obvious benefits; it has a minimal overhead of entering a password during the boot process and, if there was any additional latency in accessing the disk, it was not noticeable. The lack of any available information from an encrypted disk does highlight the issues for forensic investigators (see section 2.6) where shutting down an encrypted system means that the data may become inaccessible if the password is not available.

6.9 Similarities in File Storage

In the last section the data from the fingerprinting was found at the start of the NTFS volume and was also captured in the fingerprinting process. In this experiment the Linux installation was once again zeroed out and another fresh installation was performed with no encryption and, as before (except from setting the keyboard and region and choosing not to install office and productivity tools) the installation was the default one.

Despite the same options being checked and the same passwords being used the installation differed from the original one. It was expected that as the installation was done on a different date and at a different time this would effect some of the data, but what was not expected was that the apparent layout of the data would be that much different. One aspect that was the same (as the previous unencrypted install) was the boot partition layout showing that the differences observed between the encrypted and unencrypted installations were most likely due to the encryption option and not to the fact that the NTFS partition was occupying the second slot in the partition table as this was still the case. This time the fingerprinting was done using a Live CD in an effort to minimise any changes caused by the operation system.

Although the following images show the installation in its entirety the height of these images was reduced as these are intended to show an overview of the layout of the data and the loss of detail here is not considered important; the exceptions are those containing empty space (Figure 6.9.2 and Figure 6.8.7). The final part of the partition is also shown in more detail so that the shape of the recorded data can be seen on the NTFS partition. It is this data that will be copied to the LVM partition to see if it appears in the empty portions of this volume.

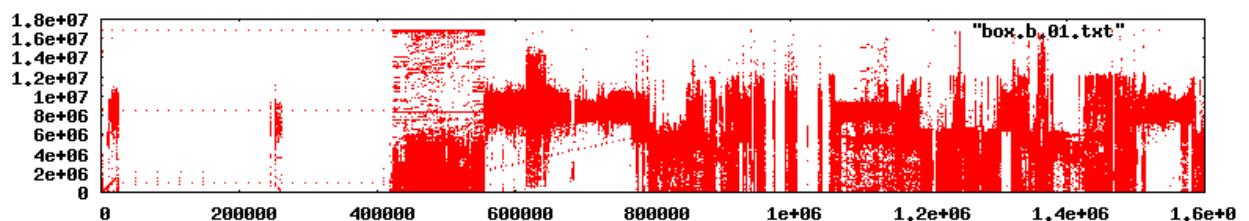


Figure 6.9.1: /boot partition and start of the LVM

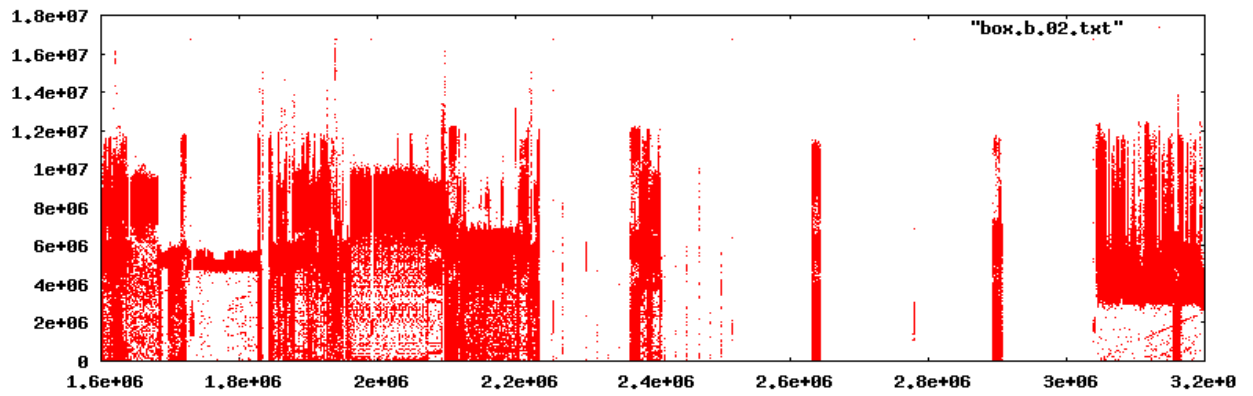


Figure 6.9.2: Part 2 of the LVM

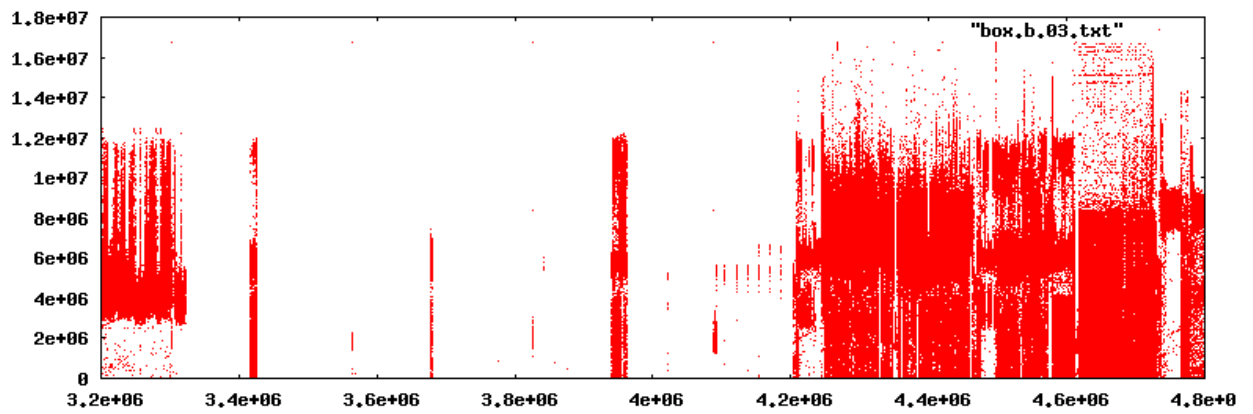


Figure 6.9.3: Part 3 of the LVM

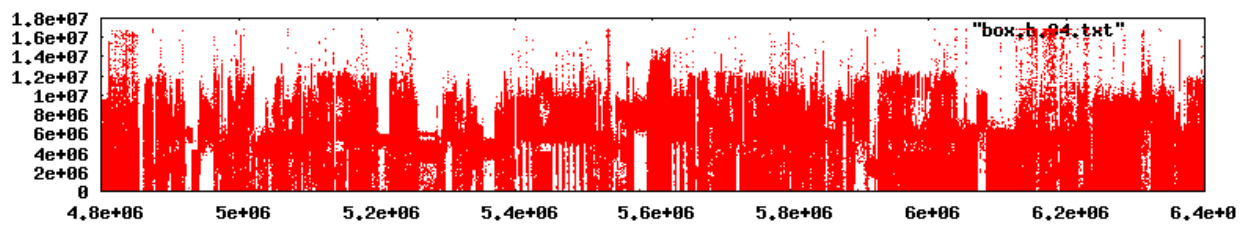


Figure 6.9.4: Part 4 of the LVM

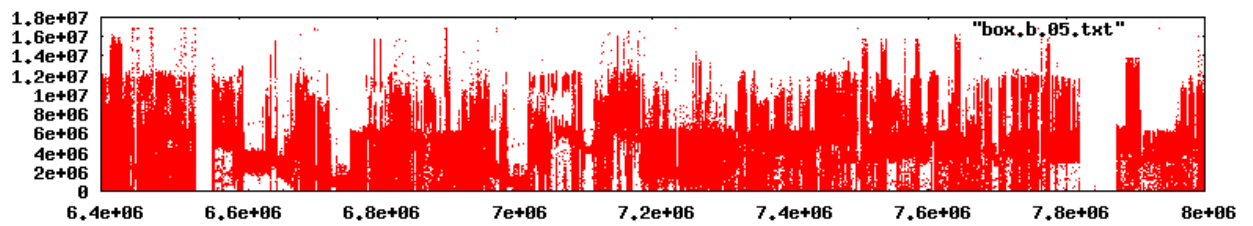


Figure 6.9.5: Part 5 of the LVM

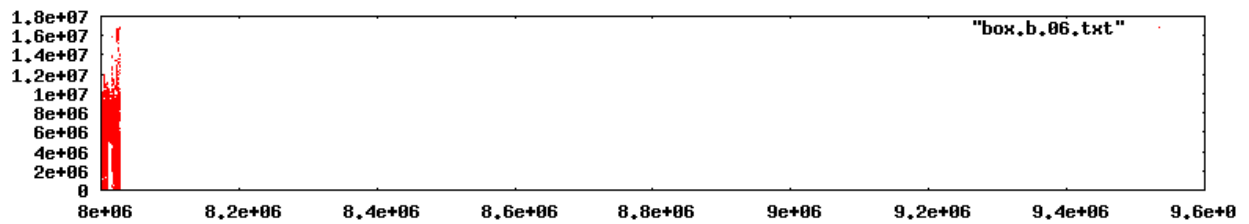


Figure 6.9.6: Part 6 of the LVM

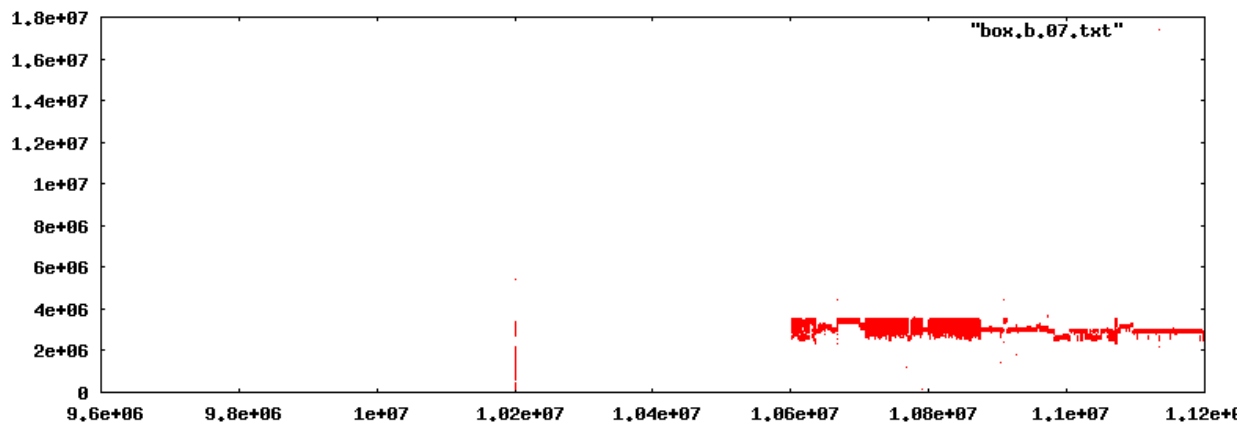


Figure 6.9.7: The end of the LVM and the start of the NTFS partition

On this installation the empty space can be found mostly in parts 2 and 3, while on the previous plain installation it was mostly in parts 4 and 5. Figure 6.9.7 is to the same scale as the previous figures and it is the data starting at about $1.06e+07$ that is of special interest here. This is the data produced by the first plain installation when it was fingerprinted. The exact contents of the data is mostly irrelevant in this context, however it is worth noting the overall shape of the plotted data. The PC was then booted to the installation and the earliest files from the NTFS were:

1. Opened using gedit and saved in ~/Documents
2. Opened in gedit and saved in ~/Download
3. Copied to ~/Music

The above directories were the first three default directories in the home Directory. Gedit is a simple text editor that comes with the default desktop. Copying the files put the disk usage from 75% to 93% and, as these were 180MB (each set), it was anticipated that they should produce noticeable changes in the fingerprint. After the files had been saved, the PC was shut down and

once again the Live CD was used to boot and iterate over the first two partitions. Although there were minor changes observed where there had previously been data, such changes should be expected as the machine had been booted using the installed operating system to do the copying. The biggest changes were where there had been empty spaces. Figure 6.9.8 and Figure 6.9.8 both show marked differences with the previous (empty) scan but more significant is that the pattern now occupying these areas of the disk is remarkably similar to that of the NTFS volume from which they were copied. The areas have been ringed to highlight which parts they are.

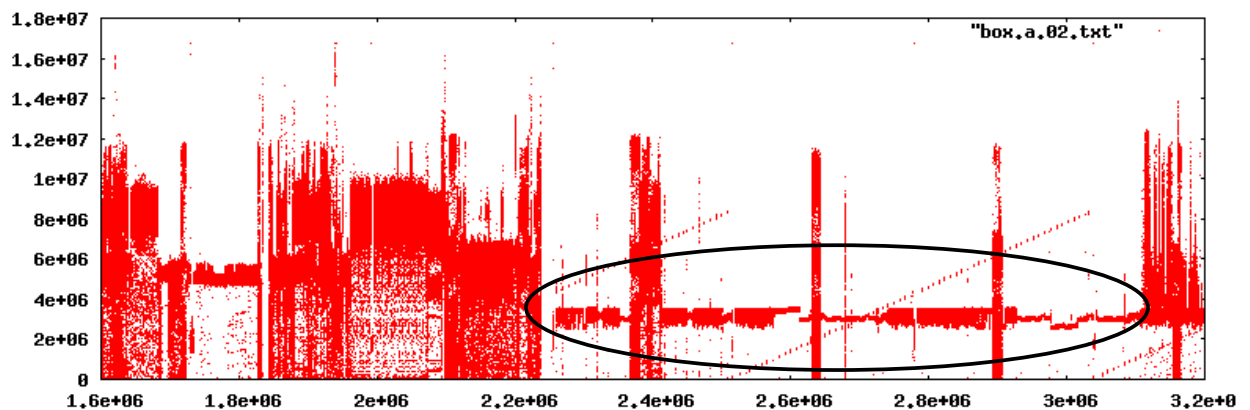


Figure 6.9.8: Part 2 of the LVM after text files copied

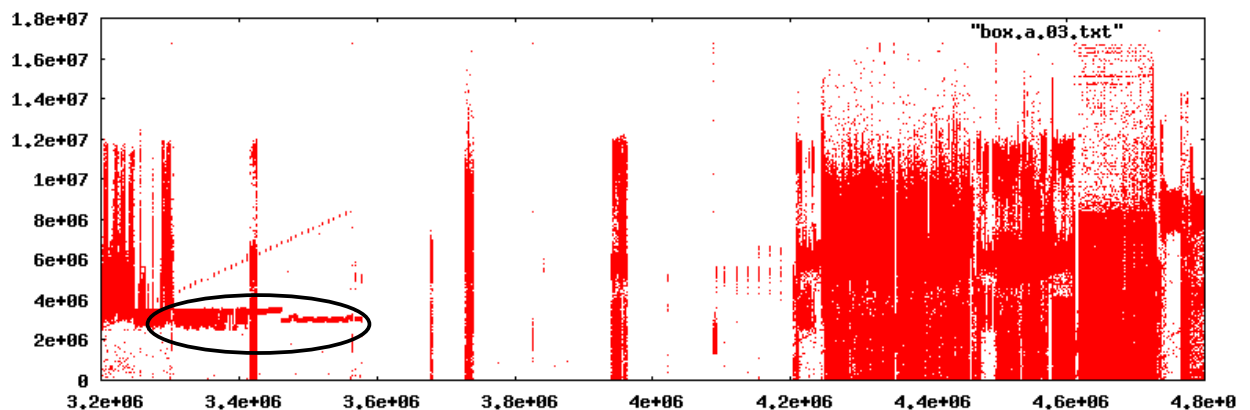


Figure 6.9.9: Part 3 of the LVM after text files copied

Sampling these areas (by using dd piped to a hexdump) showed that these areas did contain the text data previously collected.

The PC was once more booted to the installation on the hard drive and this time the command
`cat lbox.f* > bigfile.txt`

was used to write all the early files concatenated into a single large file in the home directory. By now a `df` showed the file system to be 96% full and repeating the command (output to `bigfile2.txt`) caused the output to show 100% full. The PC was shut down and once again rebooted to the Live CD and one more fingerprint taken. Figure 6.9.10 shows the result of the last copy on the text data to the installation.

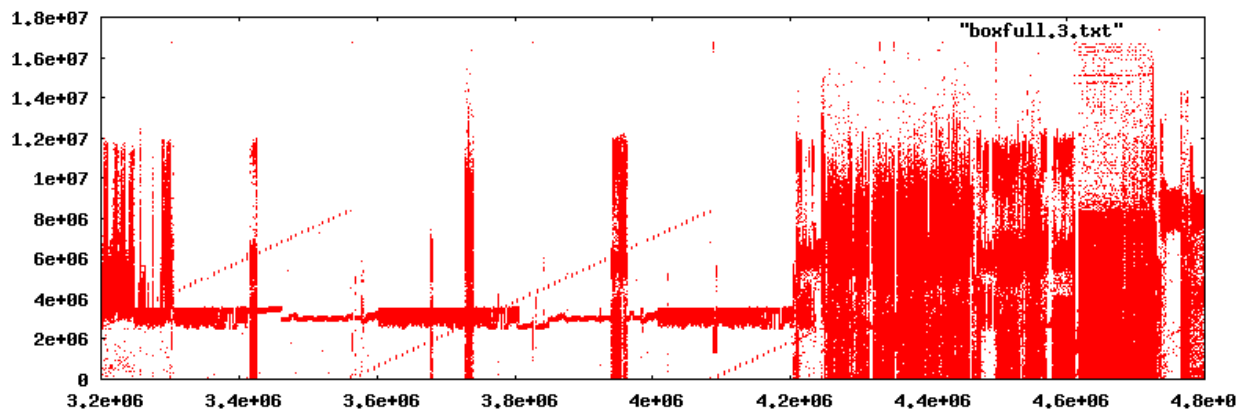


Figure 6.9.10: Fingerprint text data on part 3 of the LVM

Looking at the images of the stored data it was apparent that both the NTFS and the ext3 file systems had stored the data in a similar fashion and a hex dump of the relevant sectors showed that the data was written the same way on the disk. The diagonal lines appearing with the text data are believed to be inodes (Linux file system structures for accessing the data from within the operating system).

Although similar data had been recorded on a FAT 16 partition this logical drive had not been examined. When the primary master drive of the Windows 95B PC had been scanned the data was stored on a different physical drive and this drive had not been fingerprinted. This partition was then fingerprinted.

As before a Live CD of Fedora 8 was used (as before this only gave access to command prompts). The whole logical drive was scanned and it is clear that there was some pre-existing data present before the output of the fingerprinting process was written to a significant proportion of the drive. Figure 6.9.11 shows a similar pattern of data to both the NTFS and ext3 file systems. In this diagram it starts at about 450,000 sectors from the start of the drive to nearly 1,300,000. A hexdump reveals similar storage on the disk.

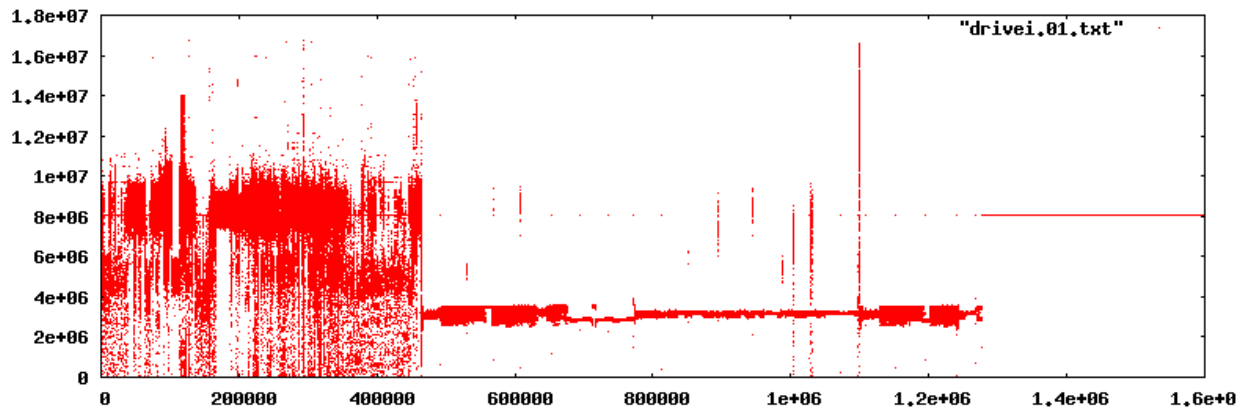


Figure 6.9.11: Drive I: containing the data from scanning of C: drives

6.10 Disk Fingerprinting Round Up

The disk fingerprinting technique used in this chapter has shown that it is possible to crawl a disk and generate an image of the data present on that disk. As pointed out in Chapter 4 the technique of summing the contents of a sector was not intended to reveal any specific result, but was simply an experiment to see what could be revealed.

Specific to the data recovery situation described in previous chapters this technique's greatest diagnostic potential was seen (as described in Chapter 5) where the location metadata was badly distorted and the fingerprint was able to produce some insights into the nature of the subsequent recording. It is strongly suspected that there will be other occasions where this tool is able to give diagnostic assistance.

In this chapter several hard drives were iterated to see what the resultant graphs would reveal. There was no expectation of discovering anything new about any of the operating systems used, the intention was purely to observe and see what was present. When looking at the fingerprints for different systems, it is important to note that unused disk drives may not all be in the same

state. In the case of disks used in the laboratory, an empty drive seemed to contain all zeros whereas on the drives in the first home PC, empty seemed to equate to a pattern of approximately half ones and half zeros. Without a full history of all the drives from manufacture to present it is not possible to determine why there are these differences in different empty states. To a large extent these differences are not important except for the fact that, when looking at a fingerprint, it must be remembered that any sector not completely full of data will tend toward the default empty condition for that disk. This means that care must be taken where comparing graphs for disks with different defaults.

When comparing the two installations of Microsoft Windows 95B it was possible to see that the patterns were broadly similar. However as the data on the disk was a collection of data (data available to the operating system, deleted data and part overwritten data), nothing more definitive could be deduced. It is suspected that a fresh installation on a disk that had zeros written to it, similar to the disk used to compare Linux installations, could provide a more definitive pattern for this operating system. While such experimentation would be feasible, the value of producing graphs for an obsolescent operating system is questionable. It was useful to be able to identify the file allocation tables and observe the changes to their contents when a drive was formatted, while the remainder of the drive was mostly untouched.

The Linux installation provided insights into the way the disk is used. Unlike the Windows installation where files appeared to be written to the start of the disk and new data took the next available empty sector(s), with Linux more of the disk is used for the installation with spaces left to store data. Such a system may make it difficult to predict where a file may be written, but when tested, it was noted that after the installation, data written to the disk filled up the empty spaces at the start of the disk before moving to the next available space.

The disk fingerprinting process showed that files are written to the hard drive in a similar way whether using NTFS, ext3 or FAT16 file systems. That is to say that the data on the disk is fundamentally the same irrespective of the file system. The exception to this is when the partition is encrypted and then very little can be determined about the system at all. The graphs of the

encrypted disk clearly show why forensic investigators must use a different approach when dealing with encrypted disks (Casey, Stellatos 2008).

7 Summary and Conclusions

7.1 Introduction

This Chapter concludes this research.

First is a brief overview of the work carried out in solving the problem (7.2) before considering what has been achieved by this research (7.3). The results of this work are then evaluated in the context of other work, specifically speculating how this research will forward the development of the Recovery Studio and how the disk fingerprinting process could be used (7.4).

7.2 Summary

Most data recovery is fairly simple. Data is stored on the recording medium and accessed by the file system. Unless there is physical damage to the medium or the data has been overwritten then recovering inaccessible data is a process of either repairing the file system so that the data can be accessed in the usual fashion or physically locating the data on the disk and copying it back into a regular file.

This research looked at a system where there was no file system and data was recorded directly to disk. In this system audio data from one telephone conversation is multiplexed with audio data from other simultaneous conversations. The only way to replay a conversation is to use its metadata to locate and extract the relevant audio segments. Normally this is not a problem and the recorded information can be replayed, however when anomalies or failures occur then reconstruction involves not only the main body of data but also the metadata and the relative offsets between the required metadata files.

The first step of a recovery job is to locate the data and metadata. This is done by reading the location metadata. Usually this information is correct, but if it is corrupted, missing or incorrect then the recovery process must wait until it is corrected before a successful outcome is possible.

The focus of this research was to find a way of verifying or recreating the location metadata.

In the initial stage the understanding of the problem proved to be incomplete and at that point it was believed that the location metadata would always be fundamentally correct with only a limited set of errors possible. On this basis a means of displaying these errors in an at-a-glance representation that could be understood by both an expert and a novice engineer was developed. The resultant representation was successful in highlighting the recognised set of errors, but by the time this artefact was completed a greater understanding of the problem permitted the realisation that the underlying heuristics would not detect all possible errors. This stage produced the at-a-glance image and a more detailed understanding of the problem.

The second stage was an attempt to recreate the location metadata by searching the recording medium to find the location of the data and metadata that had been recorded. Using several dedicated scripts (as well as a lot of human intervention) it was possible to discover the layout and make the repair in question. Attempts to automate these processes failed as bursts of corruption could be larger than a region of data. A less specific process was developed that was able to indicate the contents of any disk. Although potentially very useful, this disk fingerprinting technique could not reliably determine the location metadata used to record the current data as it could not distinguish between current and previous recordings. This stage produced the disk fingerprinting technique, a new way of looking at the recording medium and potentially diagnosing faults.

The third stage is represented by months of exploring the recording application's source code. Although this exploration did not reveal the medium apportioning formula, it produced an understanding of the nature of the methods used. This permitted a functioning formula to be deduced by working with historical data from previous repairs through trial and error. For as long as no example is encountered that does not follow the formula the problem of location of the data and metadata may be considered solved. The formula was then used in combination with the at-a-glance image process to produce an enhanced artefact able to detect and display all errors in the location metadata and generate corrected information. This stage rediscovered the formula by which the recording medium was apportioned and produced a greatly enhanced at-a-glance image.

With the main objective achieved it was possible to take a second look at the disk fingerprinting process. This process had proved insightful to repairs inside the laboratory and it was now possible to investigate its wider potential. In chapter 6 an exploration of several hard drives and operating systems was undertaken. Here it was discovered that different disks may have different states for “empty” (i.e. without data present); differing operating systems use the disk in different fashions; different file systems may have different patterns, but data stored on three different file systems was written to disk in a similar way except where the disk was encrypted. This section also looked at a simple disk format and showed what changed and what remained the same. Although no new information was uncovered, this section showed that the fingerprinting process generated a new way of looking at stored data.

7.3 Achievements

The central objective for this work was to find a technique that could verify, correct or recreate location metadata. This objective was achieved by recreating the underlying formula by which the recording medium was apportioned. It is believed that the discovered formula has been adopted for daily commercial use.

A secondary objective to find a way of displaying the data and metadata in a way that could be understood by both expert data recovery engineers and novice front-line engineering support was also achieved. Not only was an at-a-glance image developed to convey this information, but by incorporating the rediscovered formula into the image processing, errors could be detected, their presence highlighted and correct information offered.

Although not a primary objective, the development of the disk fingerprinting process has enabled a new way of looking at the contents of recording media. This technique has already given insights into one repair and provided background information on others in the laboratory, while

providing interesting insights into other systems outside of this recovery situation. It is suspected that more uses for this tool will be discovered in future. Where data recovery has been performed on disks not included in this research, it has provided some insights into disk usage that have assisted in the recovery of valuable (to the owner) data.

7.4 Enabled and Future work

Although the work carried out in this research has immediate benefits for the data recovery engineers currently working on failed systems, there may be greater long term benefit as the verification of the location metadata allows an extension of the automated processes used in the initial analysis.

The current automated process collects operational configuration files from the application partition, reads the location metadata and then copies some of the metadata regions into files. The process then halts to allow the data recovery engineer to verify that no noticeable errors have been encountered. With the rediscovered formula the automated process does not need to stop here. Assuming that at least one configuration file can be read and decoded from the application partition then the user defined field size, the number of tape drives and their type can be extracted. This information can be supplemented with the data from the `sg_readcap` utility to detect the size of the recording medium and used to verify the location metadata. If this information is successfully validated the collection of the metadata into files can proceed as it currently does, but now a greater number of metadata files can be captured and secondary processing of this metadata can be performed before any human intervention is required. This research has enabled an extension of the automation in the initial analysis stages of the recovery process.

The Recovery Studio has yet to be developed and while basic requirements exist much of the design has yet to be finalised. Although the work done in this research has provided the means of

solving two of the requirements, care needed to be exercised to ensure that as many options remain open to the eventual developer. It is to this end that all the development work has been directed toward proof-of-concept applications and not to creating polished finished products. The eventual development of the Recovery Studio is future work that falls outside the scope of this research.

(It is possible that the cost of an expert developer's time to create and test such an application may exceed the money saved by enabling less specialised engineers to perform a recovery. The reliability of the recorder means that the data recovery process is seldom required. If the recorder in question is superseded by newer products then there would be very little return for such development.)

Exploration of media using the disk fingerprinting process has proved to be very informative and it is difficult to envisage any data recovery job that would not be made easier if it was possible to look at the medium's fingerprint first. The basic concept of adding all the bits in a sector and plotting them in a graph of sum against sector is believed to be new. The technique of using hexdump to give the decimal representations was an arbitrary choice taken purely as an experiment. Ideally this process should have been tested to see if there were better ways of performing this step, but once the first graph had been produced and used, it would not have been possible to compare subsequent graphs if any changes were made to the way the script worked. This meant that while the process may not be optimal any refinements must be considered future work if comparisons between graphs already obtained were to be scientifically meaningful.

7.5 Contributions

The first contribution is the rediscovered formula by which the recorder's application allocates the recording medium. Despite limited application it is believed that this is already providing greater confidence in the initial stages of each current data recovery task. This and the developed at-a-

glance image both move the potential development of the Recovery Studio forward.

The main contribution is the development of the disk fingerprinting technique. This has already provided useful insight into at least one recording system data recovery problem and is expected to have future use in this area. Outside the specialist repair the technique is able to provide insights about other recorded media. In a general data recovery situation, this technique can provide (among other things) information about disk usage allowing recovery attempts to be focused on regions of the disk that are known to be in use.

The fingerprinting process could also be useful in a forensic investigation as it can examine the entire disk and not just the portions accessible through the normal file system and drive partitioning. If concealed areas of data existed outside these regions then a fingerprint would reveal them; however it is considered more likely that disk encryption would be used by someone wishing to hide data from an investigator (Berghel 2007).

Appendices

Appendix A –Data Storage

Although throughout this research the inference has been that data is stored on a hard drive as a collection of ones and zeros that directly relate to the information stored in the files, strictly speaking this not correct. If a hard drive is examined with a hex editor it will appear as if the underlying ones and zeros are present as expected, however, the drive contains a translation layer and when data is recorded to the magnetic medium it is not stored as ones and zeros that the user would recognise, instead the data may be a series of NRZ (Non Return to Zero) transitions where a transition may represent a one and no transition a zero. To prevent baseline wander these may be bit-stuffed or RLL (Run Length Limited). Parity coding may be added in the form of ECC (Error Correction Code). Further complexity is added by the fact the drive may map one physical area to another to avoid manufacturing defects (P-list defects) or defects that occur during the life cycle of the drive (G-list defects) (Sobey 2006).

Appendix B – Description and Source Code of Sample Puzzle

This sample puzzle is included to give some idea of the diverse nature of the requirements for the Recovery Studio. Here there was a requirement for a program that worked on a doubly linked list in the form of:

this value	next value	previous value
x	12340004	12340002
x+1	12340006	12340003
x+2 ...		

Where the objective was to find the value of x. The values in the first column increment by one each time. In this example it can be seen that x is equal to 12340003 because there is only one number between the next and the previous values in the first row and this must be the value of x. But if there are no next and previous values only two increments apart the problem becomes more interesting.

This formula was solved (by the author) by identifying applying the following rules.

	this	next	previous
	X+?	Y	Z
n lines....		Y+?	W
m lines....		W+?	V
			Y

The value of ? is the offset and is constant so $X - X+? = Y - Y+?$

$$X - X+? = Y+? - Y = n$$

$$Y - Y+? = W+? - W = m$$

$$n + m + 1 = W+? - X+?$$

The full code of the solution is in Appendix B. Basically the program iterates over the lines until the value in the next position appears in the previous position (in no more that next – previous -1 lines) and then the intermediate lines are tested to see if they pass all the preconditions. In common with many of the little applications the program accepts a simple command line argument and the output is piped to another program for more processing.

The program is used in the form:

```
./findindexinum.pl indexfilename | uniq -c
```

Where findindexinum.pl is the program name, indexfilename is the name of the input file containing the linked list information and the output is piped to `uniq` a Unix utility that counts the number of times a value occurs in a sequence. As there may be more than one lap in the metadata file, a reset or reconfiguration, more that one value may have been used over the period of recording. Typically there will be between one and three values on a disk. In the case of only one result then this information could be passed to another program and further processing could take place without additional checking. Some values such as zero are indicative of a specific problem.

```
#!/usr/bin/perl
use strict;
```

```
# This program opens the file name supplied as command line argument
# First it looks to find a line containing a sensible NextINm and PrevINum
# (though Prev not important - nice to have sanity check)
```

```
# Once a good line is found subsequent lines are stored in a hash (of arrays)
# until the NextINum value appears in the Previous.
# The hash is then iterated to determine an offset that will work for a value
# of INum - this is printed out. The program then looks for
# the next sensible NextINum and PrevINum line and repeats. Not all lines are
# evaluated - only one per NextINm - PrevINum.
```

```
# X' = X - offset (etc)          the rules
# X',Y,Z                        X < Y, X > Z
# n ----
# Y',W,X                        X - X' = Y' - Y
# m ----
# W',V,Y                        Y - Y' = W' - W
```

```
my $infile;
my $IN;
```

```
my $i;
my $y;
```

```
my $startcount;
my $offset;
my $lastoffset = 0;
my $working = 0;
my %hash = ();
my $maxlook;
```

```
if ($#ARGV != 0) {
    print " Usage = $0 < indexfilename >\n";
    exit;
}
else {
    $infile = $ARGV[0];
}
```

```
open ($IN,"$infile");
```

```
while (<$IN>) {
```

```
    #don't care about the first line
    if ($_ =~ /INum/) {
        next ;
    }
```

```
    #regex to match INum,NextINum,PrevINum
    m/(\d*),(\-?\d*),(\-?\d*)/;
```

```
    #if we have a good start then build up the hash
    if ($working == 1 ) {
        $hash {$1} = [$2, $3];
```

```
        # if the NextInum doesn't appear as a previous
        # in the max poss scope
        # then give up with this attempt and try new values
```

```

if ($1 - $startcount > $maxlook) {
    $working = 0;
    %hash = ();
}

# found the line where previous is next
if ($3 == $y) {

    # for loop till the rules are matched
    for ($i = $startcount + 1; $i < $1; $i ++) {
        if (($y - $i == $hash{$i}[1] - $startcount) &&
            ($hash{$i}[0] - $1 == $hash{$1}[1] - $i) && ($hash{$1}[1] ==
                $hash{$startcount}[0])) {
            $offset=$y - $i;

            # if we have a match and it's the same as
            # the last one escape the loop
            if ($offset == $lastoffset) {
                $i = $1;
            }
        }
    }
    $lastoffset = $offset;
    print "$offset\n";
    $working = 0;
    %hash = ();
}

}

# do we have a previous and next call and are we ready to start?
if (($2 > 0) && ( $3 > 0) && ($working == 0)){

    $y=$2;
    $maxlook = $2 - $3;
    $startcount = $1;
    #set a flag
    $working = 1;
    # start hash of annomous arrays derived
    # from scalars from regex
    $hash {$1} = [$2, $3];
}

}

close($IN);

```

Appendix C – Source Code for SVG Generation

The following code was used to generate the final SVG images as depicted in chapter 3.

```
#!/usr/bin/perl
```

```

use strict;
use Switch;

# set input and output files
my $outfile = "abslsr.svg";
my $infile = "abslsr.txt";
my $IN;
my $OUT;

# Create a hash of names and colours. This only caters for up to two tape
# drives at this time.
my %colourfile = (
    "dir.info"      => "rgb(24,116,205)",
    "diskld.dat"   => "rgb(90,144,255)",
    "disklf.dat"   => "rgb(132,142,255)",
    "oops.dat"     => "rgb(0,0,0)",
    "diskli.dat"   => "rgb(70,130,180)",
    "disklu.dat"   => "rgb(0,206,209)",
    "eventsx.dat"  => "rgb(143,188,143)",
    "cache.dat"    => "rgb(85,107,47)",
    "drivelld.dat" => "rgb(90,144,255)",
    "drivelvf.dat" => "rgb(132,142,255)",
    "drivelidat"  => "rgb(70,130,180)",
    "driveludat"  => "rgb(0,206,209)",
    "drive2ld.dat" => "rgb(90,144,255)",
    "drive2vf.dat" => "rgb(132,142,255)",
    "drive2idat"  => "rgb(70,130,180)",
    "drive2udat"  => "rgb(0,206,209)",
    "space.wst"   => "rgb(188,143,143)",
    "disk.dat"    => "rgb(139,69,19)" );

my $handyvar;

my $wrongcolour = "rgb(255,0,0)";
my $wrongborder = ";stroke-width:2;stroke:rgb(255,0,0)\"/>\n";

my $height = 30;
my $width;
my $separator = 3;
my $xrect = 135; #const
my $xtext = 10; #const
my $yrect = 10;
my $ytext = 20;

my $arg;

# svg variables
my $svgwidth; # 0.7 $viewbwidth;
my $svgheight; # 0.7 $viewbheight;
my $viewbwidth;
my $viewbheight;
my $header;

my @array;
my @outerarray;
my @errorarray;

```

```

my $i = 0;

# test the input file exists before opening this and the output file
if (-e "$infile") {
    open ($IN,"$infile");
    open ($OUT,">$outfile");
    while (<$IN>) {
        chomp;
        s/(\s)+/ /g;
        @array=split(/\s/, $_);
        # ditch the header line
        if ($array[0] eq "name") {
            next;
        }

        # handle the space at the beginning of dir.info
        if ($array[0] eq ""){
            shift(@array);
        }

        for (my $j = 0; $j < @array; $j++) {
            $outerarray[$i][$j] = @array[$j];
        }

        $i++;
    }
    close $IN;
}
else {
    print "forgot the input\n";
}

for ($i = 0; $i < @outerarray; $i++) {

    if ($outerarray[$i][0] =~ /[xd]\.dat/) {
        $yrect +=5 ;
    }
    if ($outerarray[$i][0] =~ /\.\wst/) {
        $yrect +=4 ;
    }
    $ytext = $yrect + ( $height/ 2) + 10;

    $outerarray[$i][5]= "<text x=\"$xttext\" y=\"$ytext\" font-size=\"20\" >
                        $outerarray[$i][0]</text>\n";
    $handyvar = 80 *(log($outerarray[$i][4])/log(10)) ;
    $outerarray[$i][6]= "<rect x=\"$xrect\" y=\"$yrect\" width=\"$handyvar\"
                        height=\"$height\" style=\"fill:\";
    $outerarray[$i][7]= $colourfile{$outerarray[$i][0]};#"rgb(0,0,0)";
    $outerarray[$i][8] = ";stroke-width:1;stroke:rgb(0,0,0)\"/>\n";

    if ($handyvar + $xrect > $svgwidth) {

```

```

        $svgwidth = $handyvar + $xrect;
    }
    $yrect = $yrect + $height + $separator;
    $svgheight = $yrect + 10;
}

# this is the array used to create the image
# 0 = name
# 1 = bytes
# 2 = offset
# 3 = sectors
# 4 = records
# 5 = text string
# 6 = rect
# 7 = colour
# 8 = border

my $fat;
my $blocks;
my $indexes;

for ($i = 0; $i < @outerarray; $i++) {

    if (($i > 1) && ($outerarray[$i][2] != $outerarray[$i-1][2] +
        $outerarray[$i-1][3])) {
        $outerarray[$i][8] = $wrongborder ;
        $errorarray[@errorarray] = "$outerarray[$i][0] - start bad";
    }

    switch ($outerarray[$i][0]) {

        case "dir.info"    {
            if (($outerarray[$i][1] != 4096) && ($outerarray[$i]
                [3] != 8)) {
                $outerarray[$i][7] = $wrongcolour;
                $errorarray[@errorarray] = "$outerarray[$i][0] -
                    size bad";
            }
        }
        case "diskld.dat"  {
            if ($outerarray[$i][1] != 2000) {
                $outerarray[$i][7] = $wrongcolour;
                $errorarray[@errorarray] = "$outerarray[$i][0] -
                    size bad";
            }
        }
        case "disklf.dat"  {
            $blocks=$outerarray[$i][4];
            $fat = $i;
        }
        case "diskli.dat"  {
            if ($outerarray[$i][1]%48 > 0){
                $outerarray[$i][7] = $wrongcolour;
                $errorarray[@errorarray] = "$outerarray[$i][0] -
                    non-integral records";
            }
        }
    }
}

```



```

    }
    $indexes = $outerarray[$i][4];
}
case "disklu.dat" {
    if ($outerarray[$i][4] != $indexes){
        $outerarray[$i][7] = $wrongcolour;
        $outerarray[$i-1][8] = $wrongborder;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
            non-match with indexes";
    }
}
case "eventsx.dat" {
    if ($outerarray[$i][4] != 65536){
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
            wrong size";
    }
}
case "driveld.dat" {
    if ($outerarray[$i][1] != 2000){
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
            wrong size";
    }
}
case "drivelf.dat" {
    if (($outerarray[$i][1] !=25165824) &&
        ($outerarray[$i][1] != 41943040)) {
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
            wrong size";
    }
}
case "driveli.dat" {
    if (($outerarray[$i][1] !=18874368) && ($outerarray[$i]
        [1] !=31457280)) {
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
            size may be bad";
    }
}
case "drivelu.dat" {
    if ($outerarray[$i][4] != $outerarray[$i-1][4]) {
        if ($outerarray[$i-1][7] == $wrongcolour) {
            if ($outerarray[$i][4] != $outerarray[$i
                +4][4]) {
                $outerarray[$i][7] = $wrongcolour;
                $errorarray[@errorarray] =
                    "$outerarray[$i][0] - size may be bad";
            }
        }
        else {
            $outerarray[$i][7] = $wrongcolour;
            $errorarray[@errorarray] =
                "$outerarray[$i][0] - size may be bad";
        }
    }
}

```

```

    }
}

case "drive2d.dat" {
    if ($outerarray[$i][1] != 2000){
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
                                     wrong size";
    }
}

case "drive2f.dat" {
    if (($outerarray[$i][1] != 25165824) &&
        ($outerarray[$i][1] != 41943040)) {
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
                                     wrong size";
    }
    if (($outerarray[$i][1] != $outerarray[$i - 4][1]) &&
        ($outerarray[$i - 4][7] != $wrongcolour)) {
        $outerarray[$i][8] = $wrongborder;
        $outerarray[$i - 4][8] = $wrongborder;
    }
}

case "drive2i.dat" {
    if (($outerarray[$i][1]!=18874368) && ($outerarray[$i]
        [1]!=31457280)) {
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
                                     size may be bad";
    }
    if ($outerarray[$i][1]!=$outerarray[$i - 4][1]) {
        $outerarray[$i][8] = $wrongborder;
        $outerarray[$i - 4][8] = $wrongborder;
        $errorarray[@errorarray] = "$outerarray[$i][0] -
                                     different to $outerarray[$i -4][0]";
    }
}

case "drive2u.dat" {
    if ($outerarray[$i][4]!= $outerarray[$i-1][4]) {
        if ($outerarray[$i-1][7] == $wrongcolour) {
            if($outerarray[$i][4]!=$outerarray[$i-4][4]) {
                $outerarray[$i][7] = $wrongcolour;
                $errorarray[@errorarray] =
                    "$outerarray[$i][0] - size may be bad";
            }
        }
        else {
            $outerarray[$i][7] = $wrongcolour;
            $errorarray[@errorarray] =
                "$outerarray[$i][0] - size may be bad";
        }
    }
}

case "space.wst" {
    if ($outerarray[$i][1]<165376) {
        $outerarray[$i][7] = $wrongcolour;
    }
}

```

```

                                $errorarray[@errorarray] = "$outerarray[$i][0] -
                                                                    size is bad";
                                }
                            }
    case "disk.dat" {
        if ($outerarray[$i][4] > $blocks) {
            $outerarray[$i][7] = $wrongcolour;
            $errorarray[@errorarray] = "$outerarray[$i][0]    size
                                                                    too big";

            $outerarray[$i][8] = $wrongborder;
            $outerarray[$fat][8] = $wrongborder;
        }
    }
}

my $errors = "";

$xmltext+=20;

for ($i = 0; $i < @errorarray; $i++) {

    $ytext = $yrect + ( $height/ 2) + 10;
    $errors = "$errors <text x=\"$xmltext\" y=\"$ytext\" font-size=\"20\"
                                                                    >$errorarray[$i]</text>\n";

    $yrect = $yrect + $height + $separator;
    $svgheight = $yrect + 10;
}

#assorted fudge
$xmlwidth += 10;
$xmlwidth = $xmlwidth;
$xmlheight = $svgheight;
$xmlwidth = 0.7 * $xmlwidth;
$xmlheight = 0.7 * $svgheight;
$header= "<?xml version=\"1.0\" standalone=\"no\"?>\n<svg width=\
        \"$xmlwidth\" height=\"$svgheight\" viewBox=\"0 0 $xmlwidth
$xmlheight\" \nxmlns=\"http://www.w3.org/2000/svg\" baseProfile=\"tiny\"
        version=\"1.2\"><title>abslsr</title>\n";

print $OUT $header;
print $OUT $errors;

for ($i = 0; $i < @outerarray; $i++) {

    print $OUT $outerarray[$i][5];
    print $OUT $outerarray[$i][6];
    print $OUT $outerarray[$i][7];
    print $OUT $outerarray[$i][8];
}

```

```
print $OUT "</svg>\n";
```

Appendix D – Source Code for Disk Iterating Script

The following code is the disk fingerprinting script.

```
#!/usr/bin/perl -w

use strict;
my $skip = 0;

while ($skip < 1000000000 ) {

    my $fp = samplesector();
    print "$skip $fp\n";
    $skip ++;

}

sub samplesector {

    my $input;
    my @array;
    my $tot = 0;
    $input=`dd if=/dev/sda skip=$skip bs=512 count=1 2>/dev/null | hexdump
                                                -dv`;

    @array = split(/\s+/, $input);

    for (my $i = 0; $i < @array; $i++) {
        if (length($array[$i]) < 6){
            $tot = $tot + $array[$i] ;
        }
    }
    return $tot;
}
```

Appendix E – Source code for Web Server Calculator

```
#!/usr/bin/perl

use strict;
use IO::Socket;
use Storable;
```

```
use POSIX ; #'WNOHANG';
use Switch;

# open the log file and enter the date/time program starts
my $LOG;
open ($LOG,"+>>logs/config.log");
print $LOG "\nConfiguration server started at ";
print $LOG `date`;

# create a socket or record failure
my $listen_sock = IO::Socket::INET->new(
    LocalPort => 8889,
    Type => SOCK_STREAM,
    Reuse => 1,
    Listen => 5
) or print $LOG "Could not open socket.\n";

# global variables
my $client;          # handle on socket
my $errstr = "";
my $kid;
my @kids;

my $disksize = 0;
my $udf = 0;
my $ddstype;
my $tapecount;
my $diskused = 722937;

my $recordsize;
my $bytesize;
my $sectorsize;
my $totsectors;
my $diskdat;
my $diskdatmax;
my $upload = 0;
my $boundary = "qqqqq";

my $stage = 0;

my $var1 = -1;
my $var2 = -1;
my $var3 = -1;
my $var4 = -1;

my $file = 0;
my $empty = 0;

my @arraygen;
my @outerarraygen;
my @errorarray;

my @array;
my @outerarray;
```

```

my $i = 0;

my $upload = 0;

# incoming connection from socket - fork or log failure
while ($client = $listen_sock->accept()) {
    my $pid = fork();
    print $LOG "Cannot fork\n" unless defined $pid;

    # parent: close the file handle and check for any dead kids
    # not very elegant but experiments show this worked reliably
    if ($pid) {
        close $client;
        while ($kid = waitpid (-1, WNOHANG )>0){}
        next;
    }

    # child handle client browser
    @kids = ();
    $client->autoflush(1);

    # read the what the browser is sending. Deal with it or summon a
    # subroutine to handle the
    # response, if the response is not recognised do nothing
    while ($_ = <$client>){

        # ditch carriage returns and line feeds from the line
        s/\r?\n//g;

        # ditch the empty lines for now - should be present and validated
        # for future development
        if ($_ eq "") {
            next;
        }

        # this will be the returned data from the browser
        if ( $_ =~ /^POST \/\ / ){
            $upload = 1;
            next;
        }

        # this will be true if a file is being uploaded
        if ($file == 1){

            # $upload will be true if the first line is plausible
            if ($_ =~ /^name\s/) {
                $upload = 1;
            }
            else {
                s/(\s)+/ /g;

                if ($_ =~ /---$boundary/ ){
                    #$file = 0;
                }
                else {
                    @array=split(/\s/, $_);
                }
            }
        }
    }
}

```

```

        if ($array[0] eq ""){
            shift(@array);
        }
        for (my $j = 0; $j < @array; $j++) {
            $outerarray[$i][$j] = @array[$j];
        }
        $i++;
    }
}

if ($upload == 1){

    # make a note of the file boundary
    if ($_ =~ /boundary\=(\-)*\/ ){
        $boundary = $';
    }

    elsif ($_ =~ /---$boundary/ ){
        # not required right now but may be wanted for future
        # development
    }

    # grab the field being returned
    elsif ($_ =~ /name\=\n(\d)\n\/) {
        $stage = $1;
    }

    elsif ($_ =~ /^(\d+)/ ) {

        switch ($stage) {
            case 1 {$var1 = $1;}
            case 2 {$var2 = $1;}
            case 3 {$var3 = $1;}
            case 4 {$var4 = $1;}
        }

    }

    # this signifies there is a file upload coming - set a flag
    elsif ($_ =~ /Content-Type\: text\/plain/) {
        $file = 1;
    }

    # end of upload data stop reading and work out what to send
    if ($_ =~ /$boundary(\-){2}/ ) {

        # bit of a cludge but if there is insufficient data
        # re-present the data entry page
        if (($var1 == -1)||($var2 == -1)||($var3 == -1)||
            ($var4 == -1)) {
            $errstr = "<font color=\\"#FF0000\ "> Missing or
                non numeric data </font><BR>";
            $_ = "GET / more info";
        }
        else {

```

```

        print $client abslsr($var1, $var2, $var3, $var4);

        if ($file == 1) {
            svggen();
            print $client "<BR> <img src=\"abslsr.svg\">";
        }

        print $client "</BODY></HTML>\r\n\r\n";
        close($listen_sock);
        exit;
    }
}

}

# if there's a "GET /" then send the data entry page

if (( $_ =~ /^GET \/\ / ) && ( $_ =~ /abslsr/ )){

my $IN;

        print $client "HTTP/1.0 200 OK\r\n";
        print $client "Cache-Control: no-cache\r\nConnection:
                                                                Close\r\n\r\n";

        open ($IN, "abslsr.svg");

        while (<$IN>) {
            print $client $_;
            print $_;
        }
        close $IN;
        print $client "\r\n\r\n";
        close($listen_sock);
        exit;
    }

    if ( $_ =~ /^GET \/\ / ){

        print $client "HTTP/1.0 200 OK\r\n";
        print $client "Connection: Close\r\n\r\n";
        print $client "<HTML><BODY><H1>Enter information</H1>";
        print $client "<pre><form method=\"POST\"
                                                                enctype=\"'multipart/form-data'>";
        print $client "Enter partition size in bytes ..<input
                                                                type=text name=n1><br>";
        print $client "Enter total udf size .....<input
                                                                type=text name=n2 size=5><br>";
        print $client "Enter DDS Type (3 or 4) .....<input
                                                                type=text name=n3 value=\"4\" size=2><br>";
        print $client "Enter Number of drives .....<input
                                                                type=text name=n4 value=\"2\" size=2><p>";
        print $client $errstr;
    }
}

```



```

        print $client "File to upload: <input type=file
                                name=upfile><br>";
        print $client "<input type=submit value=\"Work it out\">
                                </form>";

        print $client "</pre></BODY></HTML>\r\n\r\n";
        close($listen_sock);
        exit;
    }

    } # end of while reading from socket
} # end while socket

close($listen_sock);

#-----

sub spaces {

my $size = shift;
my $less = shift;
my $i;
my $spacestr = "&nbsp;";

    for ($i = 0; $i <= ($size - $less); $i ++){
        $spacestr = $spacestr."&nbsp;";
    }

    return $spacestr;
}

#-----

sub abslsr {

my $disksize = shift ; # $1;
my $udf = shift;      # $2;
my $ddstype = shift;  # $3;
my $tapecount = shift; # $4;

print "$disksize, $udf, $ddstype, $tapecount\n";

my $retstr = "";
my $totsectors = $disksize / 512 ;
my $int = 0;

    $retstr = "HTTP/1.0 200 OK\r\n";
    $retstr = $retstr."Connection: Close\r\n\r\n";
$retstr = $retstr."<HTML><HEAD></HEAD><BODY><H1>abslsr information</H1>";
$retstr = $retstr."<pre>name".spaces(25,5)."bytes".spaces(15,13).
    "start_address".spaces(15,7)."sectors".spaces(15,7)."records\n";

$retstr = $retstr."&nbsp;dir.info".spaces(20,4)."4096".spaces(15,6)."722925"
    .spaces(15,1)."8".spaces(15,4)."4096\n";

    $outerarraygen[$int][0] = " dir.info";
    $outerarraygen[$int][1] = "4096";

```

```

$outerarraygen[$int][2] = "722925";
$outerarraygen[$int][3] = "8";
$outerarraygen[$int][4] = "4096";
$int ++;

$retstr = $retstr."diskld.dat".spaces(19,4)."2000".spaces(15,6)."722933"
        .spaces(15,1)."4".spaces(15,4)."2000\n";

$outerarraygen[$int][0] = "diskld.dat";
$outerarraygen[$int][1] = "2000";
$outerarraygen[$int][2] = "722933";
$outerarraygen[$int][3] = "4";
$outerarraygen[$int][4] = "2000";
$int ++;

$disksize = $disksize / 2048 ;
$recordsize = ceil($disksize / 323) * 323;

if (($recordsize < 13107340) && ($ddstype == 4)) {
    $recordsize = 13107340;
}

if (($recordsize < 7864404) && ($ddstype == 3)) {
    $recordsize = 7864404;
}

$bytesize = $recordsize * 4;
$sectorsize = ceil($bytesize / 512) ;

$retstr = $retstr."disklf.dat".spaces(19,length($bytesize))."$bytesize"
        .spaces(15,length($diskused))."$diskused".spaces(15,length($sectorsize)).
        "$sectorsize".spaces(15,length($recordsize))."$recordsize\n";
$outerarraygen[$int][0] = "disklf.dat";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";
$outerarraygen[$int][4] = "$recordsize";
$int ++;

$diskused = $diskused + $sectorsize;

$recordsize = floor($disksize / 16);

if (($recordsize < 819200) && ($ddstype == 4)) {
    $recordsize = 819200;
}

if (($recordsize < 491520) && ($ddstype == 3)) {
    $recordsize = 491520;
}

$bytesize = $recordsize * 48;
$sectorsize = ceil($bytesize / 512) ;

$retstr = $retstr."diskli.dat".spaces(19,length($bytesize))."$bytesize"
        .spaces(15,length($diskused))."$diskused".spaces(15,length($sectorsize)).
        "$sectorsize".spaces(15,length($recordsize))."$recordsize\n";

```

```

$outerarraygen[$int][0] = "diskli.dat";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";
$outerarraygen[$int][4] = "$recordsize";
$int ++;

$diskused = $diskused + $sectorsize;

# $recordsize = $recordsize; # no change
$bytesize = $recordsize * $udf;

$sectorsize = ceil($bytesize / 512);

$retstr = $retstr."disklu.dat".spaces(19,length($bytesize))."$bytesize".
spaces(15,length($diskused))."$diskused".spaces(15,length($sectorsize))
."$sectorsize".spaces(15,length($recordsize))."$recordsize\n";
$outerarraygen[$int][0] = "disklu.dat";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";
$outerarraygen[$int][4] = "$recordsize";
$int ++;

$diskused = $diskused + $sectorsize;

$recordsize = 65536 ;
$bytesize = 1245184 ;
$sectorsize = 2432 ;

$retstr = $retstr."eventx.dat".spaces(19,length($bytesize))."$bytesize"
.spaces(15,length($diskused))."$diskused".spaces(15,length($sectorsize))
."$sectorsize".spaces(15,length($recordsize))."$recordsize\n";
$outerarraygen[$int][0] = "eventx.dat";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";
$outerarraygen[$int][4] = "$recordsize";
$int ++;

$diskused = $diskused + $sectorsize;

$recordsize = 130023424 ;
$bytesize = 130023424 ;
$sectorsize = 253952 ;

$retstr = $retstr."cache.dat".spaces(20,length($bytesize))."$bytesize".
spaces(15,length($diskused))."$diskused".spaces(15,length($sectorsize))
."$sectorsize".spaces(15,length($recordsize))."$recordsize\n";
$outerarraygen[$int][0] = "cache.dat";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";
$outerarraygen[$int][4] = "$recordsize";
$int ++;

```

```

$diskused = $diskused + $sectorsize;

for (my $i = 1; $i < $tapecount + 1 ; $i ++) {

    $recordsize = 2000 ;
    $bytesize = 2000 ;
    $sectorsize = 4 ;

    $retstr = $retstr."drive".$i."d.dat".spaces(18,length($bytesize))
        ."$bytesize".spaces(15,length($diskused))."$diskused".spaces
(15,length($sectorsize))."$sectorsize".spaces(15,length($recordsize))
        ."$recordsize\n";

    $outerarraygen[$int][0] = "drive".$i."d.dat";
    $outerarraygen[$int][1] = "$bytesize";
    $outerarraygen[$int][2] = "$diskused";
    $outerarraygen[$int][3] = "$sectorsize";
    $outerarraygen[$int][4] = "$recordsize";
    $int ++;

    $diskused = $diskused + $sectorsize;

    if ( $ddstype == 3) {
        $recordsize = 6291456 ;
        $bytesize = 25165824;
        $sectorsize = 49152;
    }
    else {
        $recordsize = 10485760 ;
        $bytesize = 41943040;
        $sectorsize = 81920;
    }

    $retstr = $retstr."drive".$i."f.dat".spaces(18,length($bytesize)).
"$bytesize".spaces(15,length($diskused))."$diskused".spaces(15,length
($sectorsize))."$sectorsize".spaces(15,length($recordsize)).
"$recordsize\n";

    $outerarraygen[$int][0] = "drive".$i."f.dat";
    $outerarraygen[$int][1] = "$bytesize";
    $outerarraygen[$int][2] = "$diskused";
    $outerarraygen[$int][3] = "$sectorsize";
    $outerarraygen[$int][4] = "$recordsize";
    $int++;

    $diskused = $diskused + $sectorsize;

    if ( $ddstype == 3) {
        $recordsize = 393216 ;
        $bytesize = 18874368;
        $sectorsize = 36864;
    }
    else {
        $recordsize = 655360;
        $bytesize = 31457280;
    }
}

```

```

        $sectorsize = 61440;
    }

    $retstr = $retstr."drive".$i."i.dat".spaces(18,length($bytesize)).
        "$bytesize".spaces(15,length($diskused))."$diskused".spaces
(15,length($sectorsize))."$sectorsize".spaces(15,length($recordsize))
        ."$recordsize\n";

    $outerarraygen[$int][0] = "drive".$i."i.dat";
    $outerarraygen[$int][1] = "$bytesize";
    $outerarraygen[$int][2] = "$diskused";
    $outerarraygen[$int][3] = "$sectorsize";
    $outerarraygen[$int][4] = "$recordsize";
    $int++;

    $diskused = $diskused + $sectorsize;

    # $recordsize = no change ;
    $bytesize = $recordsize * $udf;
    $sectorsize = ceil($bytesize / 512) ;

    $retstr = $retstr."drive".$i."u.dat".spaces(18,length($bytesize)).
        "$bytesize".spaces(15,length($diskused))."$diskused".spaces
(15,length($sectorsize))."$sectorsize".spaces(15,length($recordsize))
        ."$recordsize\n";

    $outerarraygen[$int][0] = "drive".$i."u.dat";
    $outerarraygen[$int][1] = "$bytesize";
    $outerarraygen[$int][2] = "$diskused";
    $outerarraygen[$int][3] = "$sectorsize";
    $outerarraygen[$int][4] = "$recordsize";
    $int++;

    $diskused = $diskused + $sectorsize;
}

$diskdatmax = floor($totsectors - ($diskused - 722925 )) ;
$diskdat = floor($diskdatmax / 1292) ;

$diskdat = $diskdat * 1292 ;

$sectorsize = $diskdatmax - $diskdat;
$bytesize = $sectorsize * 512;
$recordsize = $bytesize;

$retstr = $retstr."space.wst".spaces(20,length($bytesize))
        ."$bytesize".spaces(15,length($diskused))."$diskused".spaces
(15,length($sectorsize))."$sectorsize".spaces(15,length($recordsize))
        ."$recordsize\n";

$outerarraygen[$int][0] = "space.wst";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";

```

```

$outerarraygen[$int][4] = "$recordsize";
$int++;

$diskused = $diskused + $sectorsize;

$sectorsize = $diskdat;
$bytesize = $sectorsize * 512;
$recordsize = $bytesize / 2048;

$retstr = $retstr."disk.dat".spaces(21,length($bytesize))."$bytesize"
        .spaces(15,length($diskused))."$diskused".spaces(15,length
($sectorsize))."$sectorsize".spaces(15,length($recordsize))."$recordsize\n";
$outerarraygen[$int][0] = "disk.dat";
$outerarraygen[$int][1] = "$bytesize";
$outerarraygen[$int][2] = "$diskused";
$outerarraygen[$int][3] = "$sectorsize";
$outerarraygen[$int][4] = "$recordsize";
$int++;

$retstr = $retstr."</pre>"; #</BODY></HTML>\r\n\r\n";

return $retstr;
}

#-----

sub svggen {

my $outfile = "abslsr.svg";
my $OUT;

my %colourfile = (
    "dir.info"      => "rgb(24,116,205)",
    "diskld.dat"   => "rgb(90,144,255)",
    "disklf.dat"   => "rgb(132,142,255)",
    "oops.dat"     => "rgb(0,0,0)",
    "diskli.dat"   => "rgb(70,130,180)",
    "disklu.dat"   => "rgb(0,206,209)",
    "eventsx.dat"  => "rgb(143,188,143)",
    "cache.dat"    => "rgb(85,107,47)",
    "drivelld.dat" => "rgb(90,144,255)",
    "drivelvf.dat" => "rgb(132,142,255)",
    "drivelii.dat" => "rgb(70,130,180)",
    "drivelu.dat"  => "rgb(0,206,209)",
    "drive2d.dat"  => "rgb(90,144,255)",
    "drive2f.dat"  => "rgb(132,142,255)",
    "drive2i.dat"  => "rgb(70,130,180)",
    "drive2u.dat"  => "rgb(0,206,209)",
    "space.wst"    => "rgb(188,143,143)",
    "disk.dat"     => "rgb(139,69,19)" );

my $handyvar;

```

```

my $wrongcolour = "rgb(255,0,0)";
my $wrongborder = ";stroke-width:2;stroke:rgb(255,0,0)\"/>\n";

my $height = 30;
my $width;

my $separator = 3;
my $xrect = 135;   #const
my $xtext = 10;    #const
my $yrect = 10;
my $ytext = 20;

my $arg;

# svg variables
my $svgwidth;      # 0.7 $viewbwidth;
my $svgheight;     # 0.7 viewbheight;
my $viewbwidth;
my $viewbheight;
my $header ;

#my @array;
#my @outerarray;
#my @errorarray;

my $i = 0;

open ($OUT,"+>$outfile");

for ($i = 0; $i < @outerarray; $i++) {

    if ($outerarray[$i][0] =~ /[xd]\.dat/) {$yrect +=5 ;}
    if ($outerarray[$i][0] =~ /\.wst/) {$yrect +=4 ;}
    $ytext = $yrect + ( $height/ 2) + 10;

    $outerarray[$i][5]= "<text x=\"\$xtext\" y=\"\$ytext\" font-size=\"20\" >
                        $outerarray[$i][0]</text>\n";

    $handyvar = 80 *(log($outerarray[$i][4])/log(10)) ;
    $outerarray[$i][6]= "<rect x=\"\$xrect\" y=\"\$yrect\" width=\"\$handyvar\"
                        height=\"\$height\" style=\"fill:\"
    $outerarray[$i][7]= $colourfile{$outerarray[$i][0]};#"rgb(0,0,0)";
    $outerarray[$i][8] = ";stroke-width:1;stroke:rgb(0,0,0)\"/>\n";

    if ($handyvar + $xrect > $svgwidth) { $svgwidth = $handyvar + $xrect;}

    $yrect = $yrect + $height + $separator;
    $svgheight = $yrect + 10;

}

# 0 = name

```

```

# 1 = bytes
# 2 = offset
# 3 = sectors
# 4 = records
# 5 = text string
# 6 = rect
# 7 = colour
# 8 = border

#my $rollingoffset = $outerarray[0][2];
my $fat;
my $blocks;
my $indexes;

use Switch;

for ($i = 0; $i < @outerarray; $i++) {

    if ($outerarray[$i][1] != $outerarraygen[$i][1]) {
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] - Size wrong";
    }

    if ($outerarray[$i][2] != $outerarraygen[$i][2]) {
        $outerarray[$i][8] = $wrongborder;
        $errorarray[@errorarray] = "$outerarray[$i][0] - Start wrong";
    }

    if ($outerarray[$i][3] != $outerarraygen[$i][3]) {
        $outerarray[$i][8] = $wrongborder;
        $errorarray[@errorarray] = "$outerarray[$i][0] - Number of sectors
                                   wrong";
    }

    if (($outerarray[$i][4] != $outerarraygen[$i][4]) && ($outerarray[$i][4]
                                                            ne "")){
        $outerarray[$i][7] = $wrongcolour;
        $errorarray[@errorarray] = "$outerarray[$i][0] - Number of records
                                   wrong";
    }

}

print "$outerarraygen[$i][0], $outerarray[$i][0]\n";

}

my $errorsun = "";
$text+=20;

for ($i = 0; $i < @errorarray; $i++) {

    $ytext = $yrect + ( $height/ 2) + 10;
    $errorsun = "$errorsun <text x=\"$text\" y=\"$ytext\" font-size=\"20\"
                >$errorarray[$i]</text>\n";

    $yrect = $yrect + $height + $separator;
}

```



```
$svgheight = $yrect + 10;

}

#assorted fudge
$svgwidth += 10;
$viewbwidth = $svgwidth;
$viewbheight = $svgheight;
$svgwidth = 0.7 * $svgwidth;
$svgheight = 0.7 * $svgheight;
$header= "<?xml version=\"1.0\" standalone=\"no\"?>\n<svg width=\
    \"$svgwidth\" height=\"$svgheight\" viewBox=\"0 0 $viewbwidth
        $viewbheight\" \nxmlns=\"http://www.w3.org/2000/svg\"
        baseProfile=\"tiny\" version=\"1.2\"><title>abslsr</title>\n";

print $OUT $header;
print $OUT $errorsun;

for ($i = 0; $i < @outerarray; $i++) {

    print $OUT $outerarray[$i][5];
    print $OUT $outerarray[$i][6];
    print $OUT $outerarray[$i][7];
    print $OUT $outerarray[$i][8];

}

print $OUT "</svg>\n";

}
```

References

Adelstein, F., 2006. Live forensics: diagnosing your system without killing it first. *Communications of the ACM*, 49(2), pp. 63-66.

Basili, V. (2007): "The Role of Controlled Experiments in Software Engineering Research," in *Empirical Software Engineering Issues*, LNCS 4336, (Eds.), pp.33-37.

Basili, V.R., 1996. The role of experimentation in software engineering: past, current, and future, *ICSE '96: Proceedings of the 18th international conference on Software engineering, 1996*, IEEE Computer Society pp442-449.

Berghel, Hal, 2007. Hiding data, forensics, and anti-forensics. *Communications of the ACM*, 50(4), pp. 15-20.

Boyd, C. and Forster, P., 2004/2. Time and date issues in forensic computing—a case study. *Digital Investigation*, 1(1), pp. 18-23.

Berners-Lee, T., Fielding, R., and Frystyk, H. 1996. RFC 1945: Hypertext transfer protocol: HTTP/1.0.[online] Available at: <http://www.w3.org/Protocols/rfc1945/rfc1945> [Accessed 22 April 2009].

Canforaharman, G. and Penta, M.D., 2007. *New Frontiers of Reverse Engineering*, FOSE '07: 2007 Future of Software Engineering, 2007, IEEE Computer Society pp326-341.

Casey, E. and Stellatos, G.J., 2008. The impact of full disk encryption on digital forensics. *ACM*

SIGOPS Operating Systems Review, 42(3), pp. 93-98.

Chikofsky, E.J. and Cross, J.H.,II, 1990. Reverse engineering and design recovery: a taxonomy. Software, IEEE, 7(1), pp. 13-17.

Connolly, T. and Beeg, C., 2002. Database Systems: A practical Approach to Design, Implementation and Management 3rd Ed. Published by Pearson Education. Essex UK. ISBN: 0 201 70857 4

Defense Security Service, 1995. National Industrial Security Program Operating Manual (NISPOM), chapter 8: Automated Information System Security. U.S. Government Printing Office.

D, Mike, 2007. Beginners Guides: Hard Drive Data Recovery.[online] (Last updated: 12th April 2009) Available at: <http://www.pcstats.com/articleview.cfm?articleID=1139> [Accessed 21 January 2008]

Duncan, R., 1986. Advanced MS dos Programming, Redmond Washington USA, Microsoft Press. ISBN 0-914845-77-2

Farmer, D., 2001. Bring Out Your Dead. [online] Available at: <http://www.ddj.com/database/184404444> edn. Dr. Dobb's Journal.[Accessed 22 April 2009]

Fielding, R. Gettys, J. Mogul, J Frystyk, H. Masinter, L. Leach, P. Bernes-Lee,T., 1999. RFC 2616: Hypertext transfer protocol, HTTP/1.1, [online] Available at: <http://www.ietf.org/rfc/rfc2616.txt> [Accessed 21 January 2008]

Gattei, S., 2008. Karl Popper's philosophy of science : rationality without foundations. New York: Routledge. ISBN-10: 0415378311 ISBN-13: 978-0415378314

Gilbert D., 2009. The Linux SCSI Generic (sg) Driver [online] (Last updated: 11th April 2009) Available at: <http://sg.danny.cz/sg/index.html> [Accessed 16 April 2009]

Gnuplot 2009. gnuplot homepage [online] (last updated March 2009) Available at: <http://www.gnuplot.info/> [Accessed 16 April 2009]

Gomez, R.D., Adly, A.A., Mayergoyz, I.D. and Burke, E.R., 1993. Magnetic force scanning tunneling microscopy: theory and experiment. *Magnetics, IEEE Transactions on*, 29(6), pp. 2494-2499.

Gomez, R.D., Adly, A.A., Mayergoyz, I.D. and Burke, E.R., 1992. Magnetic force scanning tunneling microscope imaging of overwritten data. *Magnetics, IEEE Transactions on*, 28(5), pp. 3141-3143.

Harrison, W., 2006. A term project for a course on computer forensics. *Journal on Educational Resources in Computing.*, 6(3), pp. 6.

Hawking, S., 1988. *A Brief History of time*. London, Bantam Press ISBN:0593 015185

Ieong, R. and Leung, H., 2007. Deriving case specific live forensics investigation procedures from FORZA, SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, 2007, ACM Press pp175-180.

ISO/IEC 2006, International Standard - ISO/IEC 14764 IEEE Std, Software Engineering -- Software Life Cycle Processes – Maintenance 14764-2006, E-ISBN: 0-7381-4961-6, ISBN:

0-7381-4960-8

Jian-Gang zhu, Luo, Y. and Ding, J., 1994. Magnetic force microscopy study of edge overwrite characteristics in thin film media. *Magnetics, IEEE Transactions on*, 30(6), pp. 4242-4244.

Joukov, N., Papaxenopoulos, H. and Zadok, E., 2006. Secure deletion myths, issues, and solutions, *Conference on Computer and Communications Security: Proceedings of the second ACM workshop on Storage security and survivability, 2006*, ACM Press pp. 61-66.

Lister, ,Raymond, 2005. Mixed methods: positivists are from Mars, constructivists are from Venus. *SIGCSE Bull.*, 37(4), pp. 18-19.

March, S.T. and Smith, G.F., 1995. Design and natural science research on information technology. *Decision Support Systems*, 15(4), pp. 251-266.

National Industrial Security Program Operating Manual (DoD 5220.22-M), Dept. of Defense, 1995 [online]. Available: http://www.dss.mil/isec/nispom_195.htm [Accessed 02 February 2009]

NC State University, 2004. LabWrite Glossary [online]. Available: www.ncsu.edu/labwrite/res/res-glossary.html [Accessed 15 April 2009].

Phillips, ,B.J., Schmidt, ,C.D. and Kelly, ,D.R., 2008. Recovering data from USB flash memory sticks that have been damaged or electronically erased, *e-Forensics '08: Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop, 2008*, ICST pp1-6.

Ramos, C.S., Oliveira, K.M. and Anquetil, N., 2004. Legacy software evaluation model for outsourced maintainer. *CSMR 2004. Proceedings. Eighth European Conference on Software*

Maintenance and Reengineering CSMR 2004, pages 48–57,

Sammes, T. Jenkinson, B. 2007. *Forensic Computing A Practitioner's Guide*, 2nd ed. London, Springer ISBN-10: 1852332999, ISBN-13: 978-1852332990

Sobey, Charles H. 2004-04-14, 2004-last update, drive-independent data recovery [Homepage of ChannelScience], [Online]. Available: <http://www.actionfront.com/whitepaper/Drive-Independent%20Data%20Recovery%20Ver14Alrs.pdf> [Accessed 9th June 2007].

Sobey, C.H., Orto, L. and Sakaguchi, G., 2006. Drive-independent data recovery: the current state-of-the-art. *Magnetics*, IEEE Transactions on, 42(2), pp. 188-193.

Steven Bauer, Nissanka B. Priyantha, 2001-05-14, 2001-last update, Secure data deletion for Linux file systems [Homepage of Proceedings of the 10th USENIX Security Symposium, August 13–17, 2001, Washington, D.C., USA], [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec01/full_papers/bauer/bauer_html/index.html [Accessed 9th June 2007].

Symantec Corporation, 2009. Norton PartitionMagic™ 8.0 [online] Available at: <http://www.symantec.com/norton/partitionmagic> [Accessed 16 April 2009]

W3C 2008 Scalable Vector Graphics (SVG) Tiny 1.2 Specification: W3C Proposed Recommendation 17 November 2008 [online]. Available: <http://www.w3.org/TR/2008/PR-SVGTiny12-20081117/> [Accessed 15th April 2009]