Department of Computer Science

Zak Blacher

Cluster-Slack Retention Characteristics: A Study of the NTFS Filesystem

Master's Thesis

D2010:06

Cluster-Slack Retention Characteristics: A Study of the NTFS Filesystem

Zak Blacher

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Zak Blacher

Approved, 10th June, 2010

Advisor: Thijs Holleboom

Examiner: Donald Ross

Abstract

This paper explores the statistical properties of microfragment recovery techniques used on NTFS filesystems in the use of digital forensics. A microfragment is the remnant file-data existing in the cluster slack after this file has been overwritten. The total amount of cluster slack is related to the size distribution of the overwriting files as well as to the size of cluster. Experiments have been performed by varying the size distributions of the overwriting files as well as the cluster sizes of the partition. These results are then compared with existing analytical models.

Acknowledgements

I would like to thank my supervisors Thijs Holleboom and Johan Garcia for their support in the creation of this document. I would very much like to thank Anja Fischer for her help proofreading and formatting this document, and to Thijs for providing some of the graphics used to demonstrate these models. I would also like to thank Johan Garcia and Tomas Hall for providing the C and C++ code used to generate and count the file fragments. Additionally, I would like to thank the community at #windows on irc.freenode.net for their help and pointers in understanding and making sense of the NTFS filesystem. I would also like to thank the Microsoft Corporation (R) for the creation of the NTFS filesystem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This chapter will present a brief overview as to the goal of this thesis as well as provide an introduction to the FIVES project and the units used within this document.

The purpose of this dissertation is to explore and quantify the results that come from the microfragment analysis of an NTFS volume. If these results prove consistent and reliable, then the use of tail slack inspection (explained in Chapter 2) can be seen as a viable means of forensic file fingerprint recovery.

This document will perform a series of microfragment analysis tests (expanded upon in Chapter 3) and compare the results with the models described in reference [1], 'Fragment Retention Characteristics in Slack Space.' Two tests have been additionally designed to compare the matching abilities of microfragment analysis with that of *rolling hash* block recovery, a more traditional approach used in digital forensics.

1.2 F.I.V.E.S.

The goal of the Forensic Image and Video Examination Support project[2] is to develop a set of automated investigative tools to be used in conjunction with law enforcement agencies to assist in the detection of data.

The role of microfragment matching is useful in demonstrating the previous existence of offending files on a device, and is the central aspect of the FIVES toolkit. The basis of the experiments performed in Chapter 3 will be the demonstration of the precision and effectiveness of these tools.

FIVES is a targeted project within the Safer Internet Program[3].

1.3 Units

The standard block size in this document is 512 x 8 bit bytes. All units measured refer to the IEC binary unit unless otherwise specified. Efforts have been made to use the notation kibibyte (2^{10} bytes) and its abbreviation 'KiB' instead of the SI defined kilobyte (10^3 bytes) and it's respective abbreviation 'Kb'. Multiples of the IEC unit are mebibytes (MiB) and gibibites (GiB) which are 2^{20} and 2^{30} bytes respectively. More about these units can be found on the Wikipedia entry page for kilobyte[4].

This notation has not yet met widespread acceptance into the technical vernacular, and a few different standards are accepted in various computer related fields. However, in this dissertation it is important to differentiate between the SI and the IEC units as certain calculations are performed using numbers from both bases.

Chapter 2

Background

2.1 Introduction

The purpose of this chapter is to expand upon the motivation behind this thesis; the analysis of distributed files and detection of remnant data. To this end, this chapter will briefly introduce the concept of digital forensics, provide a basic overview of traditional hard drive construction and describe the process in which microfragments are generated. Furthermore, it will provide a comprehensive description of the NT filesystem features, a graphic demonstrating the general trend for file sizes on a typical NTFS partition, and finally a restatement of the formulas for calculating the expected appearance of microfragments on a device.

2.2 Digital Forensics

In contrast with criminal forensics, the goal of digital forensics is to explain the state of the digital artifact rather than determine the cause.

The current implementations of forensic file recovery involve taking a full snapshot of a hard disk or device, and performing an analysis on either the unallocated area or on the full volume. The goal of these methods is the attempt to recover metadata and unreallocated sectors from deleted files. From the deleted sectors, it is possible to extract parts of the underlying data, but the idea of fully recovering overwritten data is infeasible, as the magnetic media does not retain any history.

2.3 Hard Drive Structure

Traditionally, storage has been expressed in terms of cylinder, head, and sector count tuples (CHS). A typical hard drive is composed of several rotating platters. Each face of each platter is divided into concentric rings called cylinders. These cylinders are further divided into arc-sections called blocks, and these blocks typically store 512 bytes of data.

Each face of the each platter has a separate read head that floats just above the surface. The read seeks to a cylinder, and captures data from the chosen block. Often these devices would capture many blocks at a time (see figure 2.1.

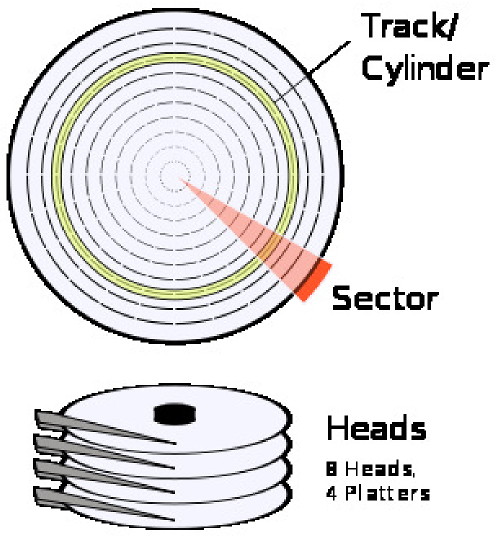

Figure 2.1: Hard Drive Physical Structure

For example, a floppy disk reports 80 cylinders, 2 heads, and 18 sectors of 512 bytes each[1]. $80 * 2 * 18 * 512\text{B} = 1474560\text{B} = 1440$ KiB, which is the standard capacity for a high-density floppy diskette[2].

This notation is not without problems, however. The original Master Boot Record specifications allowed 16 bits of information to represent 1024 cylinders, 255 heads, and 64 sectors[5], limiting devices to approximately 8.4 GiB. CHS was eventually phased out in favour of Logical Block Addressing (LBA), but most I/O devices can still report information in a CHS tuple. The more modern devices use more than 16 bits to store this information, and will report values well outside of the maxima set originally.

For instance, a consumer SDHC card purchased in 2008 reports 122560 cylinders, 4 heads, and 16 sectors of 512 bytes[3], for a total of 3830 MiB. This total is accurate despite being a solid state device and having neither heads nor cylinders!

2.3.1 Files and File Distributions

Unsurprisingly, different files with different content and different formats will occupy different amounts space. Different types of files, however, seem to follow different distributive trends. For example, we can observe that MP3 files of the same bitrate tend to be uniformly distributed across file ranges while JPEG images of the same resolution tend to be distributed geometrically. An MP3 with a bitrate of 192 kilobits per second and a length of 3 minutes occupies approximately 4.2 megabytes. This size will vary uniformly as the length of the track varies. JPEG images of a constant resolution, say 800x600 pixels, will occupy approximately 85 kilobytes with a geometric distribution around this point, depending on image content and how the JPEG compression algorithms function.

Movies ripped from DVDs and stored in AVI containers are often dynamically encoded

[1]Data collected from the hdparm utility
[2]This is often incorrectly advertised as 1.44 MB

in such a way that the file size is 700 megabytes – the amount of available space on a blank CD. While not a perfect analogue for fixed file sizes, they are usually as close as possible. Fixed file sizes are used as a parameter for the experiments as the static size comparison model does not depend upon a random number generator.

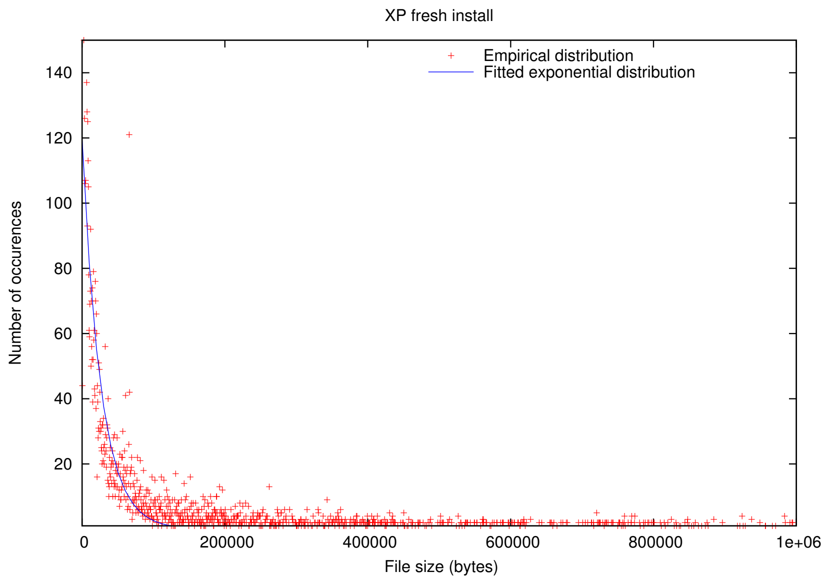

Figure 2.2: A graph of file sizes from a fresh Windows XP install

Figure 2.2 is a graph demonstrating the relative frequencies of file sizes as they occur after a fresh Windows XP install. Empirical data collected from discarded hard disks demonstrates a similar trend in file occupation on devices used for home consumption.

2.3.2 Microfragment Analysis

To ease in complexity of storing data to the hard drive and to reduce the addressing overhead, blocks are grouped together in clusters, with a cluster being the smallest individually addressable section. When a file is written to the hard disk, the filesystem drivers in the operating system create a master file table (MFT) entry, determine how many clusters are necessary for storage (rounding up for partial occupation), allocates them, and then writes the data to the device.

The data is densely packed into clusters (linearly occupying all blocks per cluster) with the exception of the final (or tail) cluster, which has only the remaining number of blocks written to it. These clusters do not need to be contiguous, and can be distributed among different tracks or platters, but are often placed as close together as possible to reduce lookup times. This is a process known as *fragmentation* and is not to be confused with the term *microfragment*

For example, in a typical NTFS filesystem with a block size of 512 bytes and a cluster size of 8 blocks (4 KiB clusters), a 10 KiB file would occupy 3 clusters, but only require 20 of the available 24 blocks. This example can be seen in Figure 2.3. In NTFS, neither clusters nor blocks are shared between different files.

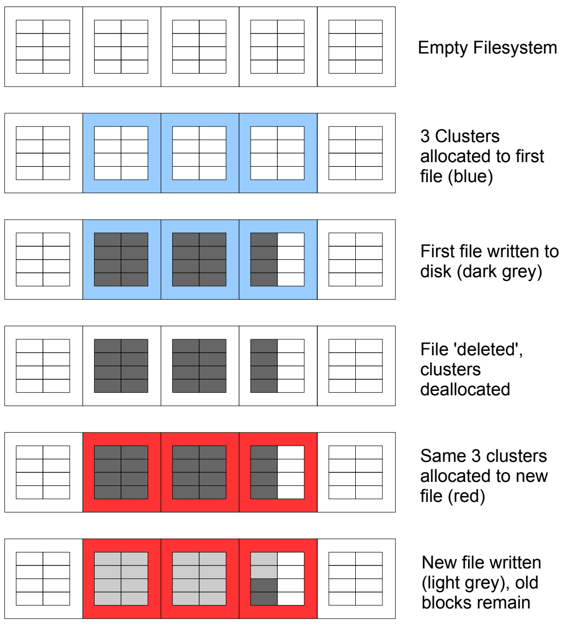
Empty Filesystem
3 Clusters allocated to first file (blue)
First file written to disk (dark grey)
File 'deleted', clusters deallocated
Same 3 clusters allocated to new file (red)
New file written (light grey), old blocks remain

Figure 2.3: A graphical representation of sub-cluster block writing

When a file is deleted, the NTFS driver only deletes the MFT entry, effectively abandoning the allocated clusters rather than removing them. This makes the undeletion process (recovery of actual data) possible, provided the blocks are not reallocated to other files. If we return to the previous example, deleting the 10 kibibyte file would deallocate the 3 occupied blocks. Writing a new file of 9 KiB to the same location would reoccupy the 3 clusters, but only 18 blocks. Not zeroing the unused blocks in the cluster is faster, and less intense on the physical hardware, but does present a security problem in which the remaining 2 blocks past the end of our new file contain data from our first file.

The analysis of these remaining 2 blocks, or file microfragment, may yield information thought deleted by the user. This paper studies the frequency and occurrences of these microfragments.

2.3.3 NTFS

2.3.3.1 Overview

The NTFS filesystem was developed by Microsoft for the release of their Windows NT operating system. NTFS supports journaling, hard links, alternate data streams (ADS), sparse files, transparent encryption and compression, volume shadow copy, and copy-on-write. (See table 2.1)

Typical formatting parameters for the filesystem are blocks of 512 bytes and clusters of 8 blocks. When a block is written to, the remaining slack within the block is zeroed out. The number of files allowed on the filesystem is essentially limited by the number of available clusters on the partition, as each cluster can only contain one file[6].

2.3.3.2 Master File Table

The NTFS Master File Table (MFT) reserves approximately 12.5%[8] of the clusters for file record entries. The MFT contains entries defining header information, specific volume

NTFS Version	1	1.1	1.2 (4.0)	3.0 (5.0)	3.1 (5.1)	3.1 (5.2)	3.1 (6.0)
Windows Release	NT 3.1	NT 3.5	NT 3.51	2000	XP	2003	Vista
Year	1993	1994	1995	2000	2001	2003	2005
Forward Compatible		X	X	X	X	X	X
FAT Long Names		X	X	X	X	X	X
Compressed Files			X	X	X	X	X
Named Streams			X	X	X	X	X
ACL Security			X	X	X	X	X
Disk Quotas				X	X	X	X
Encryption				X	X	X	X
Sparse Files				X	X	X	X
Reparse Points				X	X	X	X
USN Journaling				X	X	X	X
Expanded/Redundant MFT					X	X	X
Volume Shadow Copy					X	X	X
Persistant Snapshots						X	X
Transactional NTFS							X
Symbolic Links							X

Table 2.1: NTFS Feature List[7]
Unoffical NTFS versioning information in brackets

information (such as bad blocks or quota information), and file records. The MFT is often allocated contiguously but may grow and shrink as the demands on the filesystem change.

2.3.3.3 Records

Each file record in the MFT contains the filename and path, security descriptor, other associated metadata and either the location of the file content or the content itself, depending on the size. For larger non-resident metadata attributes such as an alternate data stream[9], a reference is stored for an extent record in the record block. Each record occupies 1024 or 4096 bytes, depending on the version, but regardless of filesystem format parameters[10].

2.3.4 Expected Microfragment Distribution

When data is written to the hard disk device the final (or tail) cluster of an allocated group contains the terminating blocks of the file. With a cluster size of 8 blocks, there would be between 1 and 8 blocks occupied by the tail end of the new file, leaving 0 to 7 available with the data remaining from a previous write. Note that a tail cluster will never contain all 8 blocks as slack, as this would imply that 0 blocks were needed from this cluster by the new file.

The actual numbers and apparent frequencies of cluster slack blocks in a file system depend strongly on the size and distribution of both overwritten and new files, as well as the characteristic parameters of the file system.

2.4 Overview of the Microfragment Analysis Model

In order to properly compare the measured results to the modelled values, we first need to restate the existing formulae found in the paper 'Fragment Retention Characteristics in Slack Space.'[1]

In the following formulae, we will use the notation C to mean cluster size (in bytes), B to mean block size (in bytes), D to mean detection area (1 gibibyte in this document), S to mean file size (bytes), and $\bar{S}$ to be average file size (also bytes).

$N_C^{(S)}$ is the number of clusters allocated to a file. As earlier discussed, this is equal to the number of blocks required (rounded up) divided by the size of a cluster, and rounded up; or more formally $\lceil \frac{S}{C} \rceil$, provided the file is large enough not to be stored directly in the MFT.

W_C is the number of end clusters with the possibility of containing microfragments, and W_R is the number of microfragments detected, having factored in P, the probability that a file will leave a microfragment.

The derivation and indepth explanation of these formulae is beyond the scope of this document, and can be found in the referenced paper.

2.4.1 Fixed Distribution

In a volume on which initial files have been generated with a constant file size of S_F, the expected microfragment population W_R should appear with the following frequency:

$$W_R^{(c)} = \frac{D}{\left\lceil \frac{S_F}{C} \right\rceil C} \tag{2.1}$$

2.4.2 Uniform Distribution

With a uniform distribution, the numbers become a little bit more complex. Rather than a fixed file size, we can say that all S lie uniformly distributed within the range $L_1 ... L_2$, allowing us to approximate the average file size of S ($\bar{S}$) to be $\frac{L_1 + L_2}{2}$ bytes.

If we define $L^{(+)C} = \left\lceil \frac{L}{C} \right\rceil C$ and $L^{(-)C} = \left\lfloor \frac{L-1}{C} \right\rfloor C$, to be respectively the largest and smallest integer multiples of C closest to L, then within the range of L_1 and L_2, we can expect that files within the range $L_1^{(-)C} ... L_2^{(+)C}$ will have an average of $\frac{C}{2B}$ slack blocks. Our tail ranges could be expected to have approximately $\frac{N_C^{(L_1)} + \frac{C}{B}}{2}$ and $\frac{N_C^{(L_2)}}{2}$ blocks in the lower and upper distribution tail ranges respectively.

We can approximate the expected microfragment recovery to be the following:

$$W_C^{(u)} = W_R^{(u)} P^{(u)} = \frac{D}{\bar{N}_C^{(u)} C} (1 - \frac{B}{C}) \tag{2.2}$$

where

$$\begin{aligned}\bar{N}_C^{(u)} = \frac{1}{C} \frac{1}{L_2 - L_1 + 1} \Big(& L_1^{(+)C} (L_1^{(+)C} - L_1 + 1) \\ & + \frac{1}{2} (L_2^{(-)C} - L_1^{(+)C})(L2^{(-)C} + L_1^{(+)C} + C) + \\ & L_2^{(+)C} (L_2 - L_2^{(-)C} \Big)\end{aligned} \tag{2.3}$$

and

$$P^{(u)} = 1 - \frac{B}{C} \tag{2.4}$$

$P^{(u)}$ is a correction factor for file sizes because the amount of cluster slack that is less than one block cannot not detected. See reference [11] for more details.

2.4.3 Exponential Distribution

Similar to the uniform distribution, we can apply the same functions and assumptions, but as we are using a geometric distribution for file, some of the averaging functions are altered slightly.

Our average file size can now be approximated by

$$S^{(e)} = \sum_{n=L_1}^{L2} n\, p_n = \frac{1}{1-e^{-b}} \times \frac{\left(L_1 - (L_1-1)e^{-b}\right) e^{-bL_1} - \left(L_2 + 1 - L_2 e^{-b}\right) e^{-b(L_2+1)}}{e^{-bL_1} - e^{-b(L_2+1)}} \tag{2.5}$$

and the average number of allocated clusters can be expressed as

$$\begin{aligned}\bar{N}_C^{(e)} = {} & \frac{L_1^{(+)C}}{C} \frac{e^{-bL_1} - e^{-b(L_1^{(+)C}+1)}}{e^{-bL_1} - e^{-b(L_2+1)}} + \\ & \frac{e^{-b}}{e^{-bL_1} - e^{-b(L_2+1)}} \times \frac{1}{1-e^{-bC}} \times \\ & \left(\left[\frac{L_1^{(+)C}}{C}(1-e^{-bC}) + 1 \right] e^{-bL_1^{(+)C}} - \right. \\ & \left. \left[\frac{L_2^{(-)C}}{C}(1-e^{-bC}) + 1 \right] e^{-bL_2^{(-)C}} \right) + \\ & \frac{L_2^{(+)C}}{C} \frac{e^{-b(L_2^{(+)C}+1)} - e^{-b(L_2+1)}}{e^{-bL_1} - e^{-b(L_2+1)}}\end{aligned} \tag{2.6}$$

Our expected recovery count can be expressed as

$$W_C^{(e)} = W_R^{(e)} \, P^{(e)} \tag{2.7}$$

where the block correction factor in this case is

$$P^{(e)} = \frac{1 - e^{-b\,(C-B)}}{1 - e^{-b\,C}} \tag{2.8}$$

See Reference [11] for the detailed derivation of these formulas.

Chapter 3

Experiments

3.1 Introduction

The purpose of the following experiments is to create filesystems in which the remaining microfragment data conforms to an expected file distribution, and then compare the collected data with that of the model. These experiments will demonstrate how different file overwriting parameters affect the number of cluster slack blocks left on the filesystem.

For the following experiments, we generate 1000 x 250 kibibyte files of known content on a 1 gibibyte partition. These files are then deleted and overwritten with randomly generated data conforming to specified file sizes, referred to as *random files* in this paper. With standard NTFS formatting, each of these 250 kibibyte files will occupy 63 x 4 kibibyte clusters with only half of the tail cluster containing data. Approximately 12.5% of the clusters on the physical volume are reserved for the master file table meaning that approximately 27.5% of the usable file system will be initially occupied by this data. After these files are deleted, the partition is filled with random files containing random data. These files occupy the previously used clusters, and the remaining slack is analyzed for the fingerprints of the initial data.

Each experiment writes files conforming to the flags on each line of the rf_params value in the parameters subsection onto the device. These experiments are performed as many times as is specified by the field 'reps'. The variations in frequencies of microfragment recovery should match the projected values.

3.1.1 Testbed

For the following experiments, I will be using a test machine running Microsoft Windows XP Home (R)with Service Pack 3 as it's operating system. Tests will be performed in an environment running Cygwin(R) version 1.7.5 and Python 2.5 and on a device with 1 gigabyte of storage. The Python scripts used for collecting and interpreting this document have been included in the appendix. The C and C++ sources as well as the raw collection data and OpenOffice (R) documents used for the generation of the graphics have been included with the offline distribution of this paper.

3.2 findGenFrag Experiments

The following two tests were performed in order to compare our predictive models to the results gathered through real-world experimentation.

3.2.1 'File Size Distribution' Test

3.2.1.1 Introduction

This experiment exists to gather data as a baseline to comparison with our existing models. We will generate files with many different file size distribution characteristics and then contrast our empirical data with our calculated results.

3.2.1.2 Experiment

Because the generation of the random content files has been set up to fill the entire device, we can expect to see a tiling effect over the filesystem. For example, files with a fixed size of 10 kibibytes would each occupy 3 clusters (12 kibibytes), leaving half a cluster of slack data in the tail. Filling the filesystem with these files would leave approximately 1/6th of the original data behind. As 250 000 kibibytes (25%) of the filesystem was previously occupied by our 1000 x 250 kibibyte files, we may expect that approximately 1/12 (1/4 * 1/3)[1] of our clusters contain slack data. The actual number will be somewhat lower as there will be no slack data remaining where the tail of our random file is written to a cluster previously containing the tail of our fixed content file.

[1](original occupation * tail frequency)

3.2.1.3 Parameters

These, and subsequent Parameters subsections define the set of random file generation parameters, as well as other environmental settings.

```
reps = 5
fs_types = ['ntfs']
cluster_size = ['4096']
of_params = [
 ('1000 files 250 Kbyte'  , ['-s', '250', '-c', '1000']),
]
rf_params = [
 ('Exponential: 10 Kbyte'     , ['-e', '10' , '0.0006']),
 ('Exponential: 20 Kbyte'     , ['-e', '20' , '0.0006']),
 ('Exponential: 40 Kbyte'     , ['-e', '40' , '0.0006']),
 ('Exponential: 80 Kbyte'     , ['-e', '80' , '0.0006']),
 ('Exponential: 141 Kbyte'    , ['-e', '141', '0.0006']),
 ('Exponential: 800 Kbyte'    , ['-e', '800', '0.0006']),
 ('Uniform: 8-12 Kbyte'       , ['-u', '8'  , '12']),
 ('Uniform: 16-24 Kbyte'      , ['-u', '16' , '24']),
 ('Uniform: 36-44 Kbyte'      , ['-u', '36' , '44']),
 ('Uniform: 76-84 Kbyte'      , ['-u', '76' , '84']),
 ('Uniform: 600-1000 Kbyte'   , ['-u', '600', '1000']),
 ('Fixed: 10 Kbyte'           , ['-s', '10']),
 ('Fixed: 20 Kbyte'           , ['-s', '20']),
 ('Fixed: 40 Kbyte'           , ['-s', '40']),
 ('Fixed: 80 Kbyte'           , ['-s', '80']),
 ('Fixed: 800 Kbyte'          , ['-s', '800']),
 ('Fixed: 8 Mbyte'            , ['-s', '8Mb']),
 ('Fixed: 80 Mbyte'           , ['-s', '80Mb']),
]
```

3.2.2 'Cluster Size Distribution' Test

3.2.2.1 Introduction

The purpose of this test is to demonstrate how different cluster sizes affect the number of slack blocks recovered. It stands to reason that the size of the cluster with respect to the size of the initial file will generate different amounts of slack data.

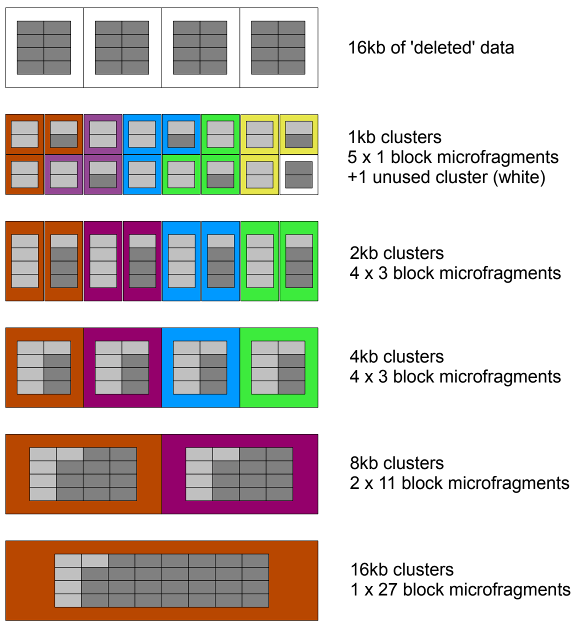
16kb of 'deleted' data
1kb clusters
5 x 1 block microfragments
+1 unused cluster (white)
2kb clusters
4 x 3 block microfragments
4kb clusters
4 x 3 block microfragments
8kb clusters
2 x 11 block microfragments
16kb clusters
1 x 27 block microfragments

Figure 3.1: Cluster Size effect on Slack Recovery

Each color represents a 2.5 kibibyte file. Light grey is new data. Dark grey is old data.

3.2.2.2 Experiment

In this test we are using a smaller set of random file parameters, but running this set against different cluster sizes to see how much resulting data can be recovered. For instance, writing uniformly distributed 4-12 kibibyte files on to 32 kibibyte clusters should leave approximately 48 blocks[2] on average in every cluster, whereas the same random file

[2] $64\text{blocks} - 2\text{blocks/kibibyte} * \frac{4+12}{2}\text{kibibytes}$

generation on clusters of 4 kibibytes will leave about 3.5[3] sectors per tail (every second cluster) on average. (see figure 3.1)

3.2.2.3 Parameters

```
reps = 5
fs_types = ['ntfs']
cluster_size = ['1024','2048','4096','8192','16k','32k']
of_params = [
 ('1000 files 250 Kbyte'  , ['-s', '250', '-c', '1000']),
]
rf_params = [
 ('Exponential: 141 Kbyte'    , ['-e', '141', '0.0006']),
 ('Exponential: 40 Kbyte'     , ['-e', '40' , '0.0006']),
 ('Exponential: 800 Kbyte'    , ['-e', '800', '0.0006']),
 ('Uniform: 10-30 Kbyte'      , ['-u', '10' , '30']),
 ('Uniform: 20-60 Kbyte'      , ['-u', '20' , '60']),
 ('Uniform: 4-12 Kbyte'       , ['-u', '4'  , '12']),
 ('Uniform: 40-120 Kbyte'     , ['-u', '40' , '120']),
 ('Uniform: 400-1200 Kbyte'   , ['-u', '400', '1200']),
]
```

[3]average expected result of a uniform distribution over the range 0 through 7

3.2.3 '30 Repetitions' Test

3.2.3.1 Introduction

In order to determine whether or not our results can be seen as statistically reliable, the following test has been designed to demonstrate the precision of our system. We will perform many repetitions of the same few tests and determine whether or not individual results with the same test parameters differ significantly.

3.2.3.2 Experiment

Because of the large amount of time[4] needed to perform each individual test, the sample of tests performed has been reduced to only four. These four tests have been selected to compare and contrast the performance of larger and smaller file size ranges versus uniform and exponential size distributions.

3.2.3.3 Parameters

```
reps = 30
fs_types = ['ntfs']
cluster_sizes = ['4096']
of_params = [
 ('1000 files 250 Kbyte'  , ['-s', '250', '-c', '1000']),
]
rf_params = [
 ('Exponential: 20 Kbyte'     , ['-e', '20' , '0.0006']),
 ('Exponential: 800 Kbyte'    , ['-e', '800', '0.0006']),
 ('Uniform: 10-30 Kbyte'      , ['-u', '10' , '30']),
 ('Uniform: 400-1200 Kbyte'   , ['-u', '400', '1200']),
```

[4]between 2-4 hours each on the given testbed, depending on generation parameters

```
]
```

3.2.4 'Rolling Hash' Tests

3.2.4.1 Introduction

The rolling hash tests use a different methodology for examining a filesystem for our targeted files. Rather than focus on blocks in tail clusters, we examine the filesystem as a whole. We perform a rolling hash calculation on a moving window that moves in 1 byte steps across the device. When our rolling hash matches a trigger value[5], we examine a logical block of 512 bytes from this point, perform a hash of this block, and compare it to our known data hashes. If this matches, we have part of an offending file. If not, we go back to our window and continue searching. A rolling hash window and a trigger value are used to reduce the amount of database lookups and increase the speed at which a volume is analyzed.

The reason we use a single byte step is that modifying data header information or compacting certain files together will alter the sub block alignment, but not the majority of the data content of the files. MP3s and JPEG images, for example, are already compressed and are not altered when put into an archive or data container object, but may be placed across block and sector boundaries as slack space is removed.

It is worth mentioning that the rolling hash recovery routines do not differentiate between the allocated state of a cluster; leading to higher recovery rates at a cost of increased scan time. This will make the comparison between microfragment recovery and hash block matching somewhat more difficult.

3.2.4.2 Experiment

We will again perform the first two tests (sections 3.2.1 and 3.2.2) using the same parameters, but analyzing the device with the rolling hash algorithm rather than simple slack analysis. This is done to compare the two methods in terms of data recovery ability.

[5] 42

The use of this approach to forensic data recovery should give us a good indicator as to the effectiveness of cluster slack forensic analysis versus traditional volume block analysis.

3.2.4.3 Parameters

(see the Parameters subsections of 3.2.1 and 3.2.2)

3.3 Summary

Our five experiments have been designed to demonstrate the effectiveness of microfragment analysis. The first and last two demonstrate physical and logical block recovery techniques respectively, and the third test demonstrates the confidence of our collection methods.

Chapter 4

Results

4.1 Introduction

The results seem to fall in line with what had been expected from the analytical model[1]. NTFS has some interesting characteristics when files of different sizes are written to it. Earlier we stated that the MFT occupies approximately 12.5% of the space on the partition, but the actual amount varies depending on the physical occupation of the usable space. For example, a device with many small files would require more space to describe and maintain attributes and metadata, and thus have a larger MFT. An extreme example of this would be an NTFS filesystem completely occupied with 1 byte files. These files are small enough to store directly in the MFT, and as such this device would have all of its space devoted to the file table. Conversely, a filesystem containing only one large file would require a single record in the master file table.

The determination of the optimal parameters in terms of filesystem construction goes beyond the scope of this paper, but was most likely a factor for determining the defaults for NTFS.

These factors, coupled with the wear of the physical medium during the strain of these tests, and the fact that theory and practice often differ all affect the actual numbers gathered.

4.2 findGenFrag Results

The following two subsections detail the results of our first two experiments.

4.2.1 'File Size Distribution' Test

4.2.1.1 Observation

For the fixed size tests in Figure 4.1, we see that where the file size was an integer multiple of the cluster size (4 kibibytes), there were virtually no microfragments remaining. This is due to the fact that the filesystem was completely overwritten by the random content files. 20 kibibyte files occupy 5 full clusters, leaving no slack data.

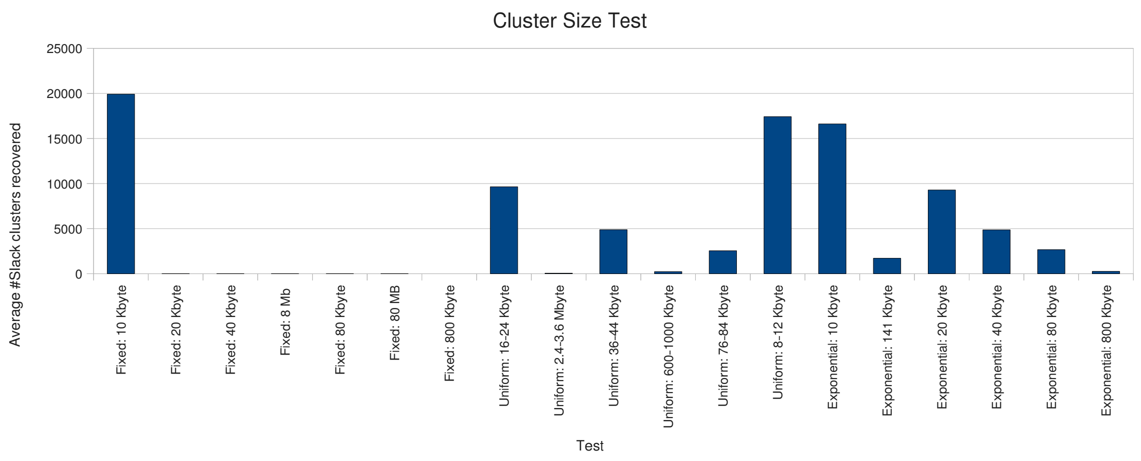

Figure 4.1: Results of Cluster Size Distribution Test

However, for our example from the previous chapter, we get an approximate average of 19917 microfragments recovered after our overwrite with 10 kibibyte files. This is less than the expected 21764 (1/12 of our original 1 gibibyte partition), but can be explained by the alignment of the tail sectors.

A 250 kibibyte file occupies 62.5 clusters but reserves 63, and a 10 kibibyte file occupies 2.5 clusters but reserves 3, meaning that every 21st 10 kibibyte file written will have it's tail over cluster containing the tail of our 250 kibibyte file. Only 5/63 (1/4 * 1/3 * 20/21) of our tail sectors can be expected to contain data from our original set. A demonstration of this can be seen in figure 4.2.

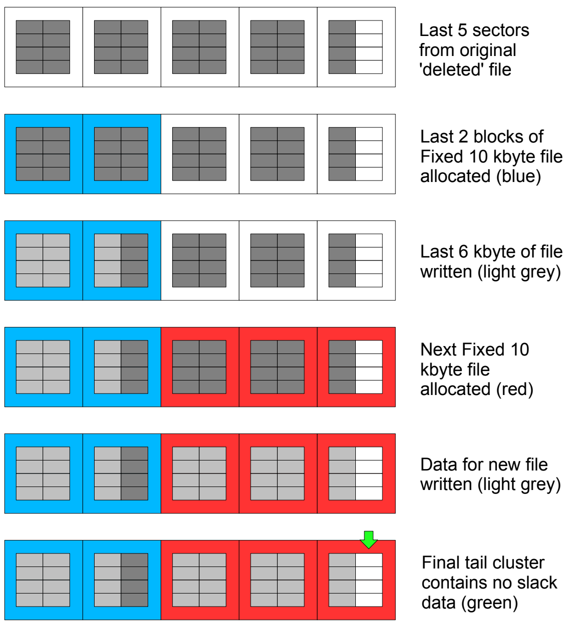
Last 5 sectors from original 'deleted' file
Last 2 blocks of Fixed 10 kbyte file allocated (blue)
Last 6 kbyte of file written (light grey)
Next Fixed 10 kbyte file allocated (red)
Data for new file written (light grey)
Final tail cluster contains no slack data (green)

Figure 4.2: Fixed 10 Kbyte files resulting in a tail overlap scenario

4.2.2 'Cluster Size Distribution' Test

4.2.2.1 Observation

Our results are fairly straightforward for this test. As the cluster size increases exponentially, the number of recovered blocks increases exponentially. From this we can see that the recovered amount of data depends more on cluster size and initial data than on the overwriting parameters. With smaller cluster sizes, less data is recovered as the amount of space left in the the tail decreases.

Figures 4.3 and 4.4 demonstrate this trend very well. In figure 4.3, we see an increasing trend in the number of recovered blocks with respect to cluster size across all random file distribution patterns and in figure 4.4 we can clearly see that the number of recoverable blocks per cluster occur at the similar ratios with respect to random file parameter.

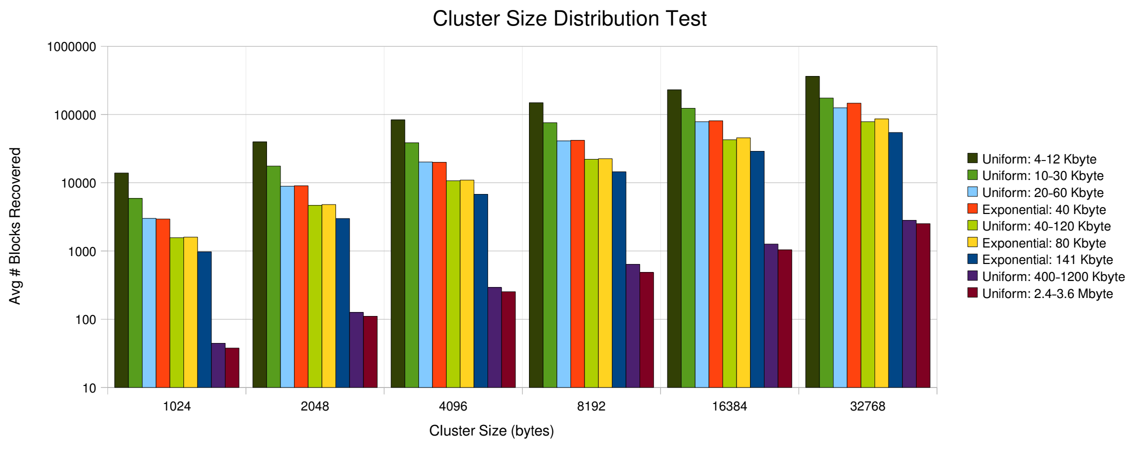

Figure 4.3: Results of Cluster Size Distribution Test (rf param v. cluster size)

Our third graph (figure 4.5) is organized slightly differently. This graph demonstrates the trends of *microfragment* recovery instead of *block* recovery. As each cluster can contain at most one microfragment, it's no surprise that as the cluster size increases and the average file size decreases, the likelihood of microfragment recovery increases.

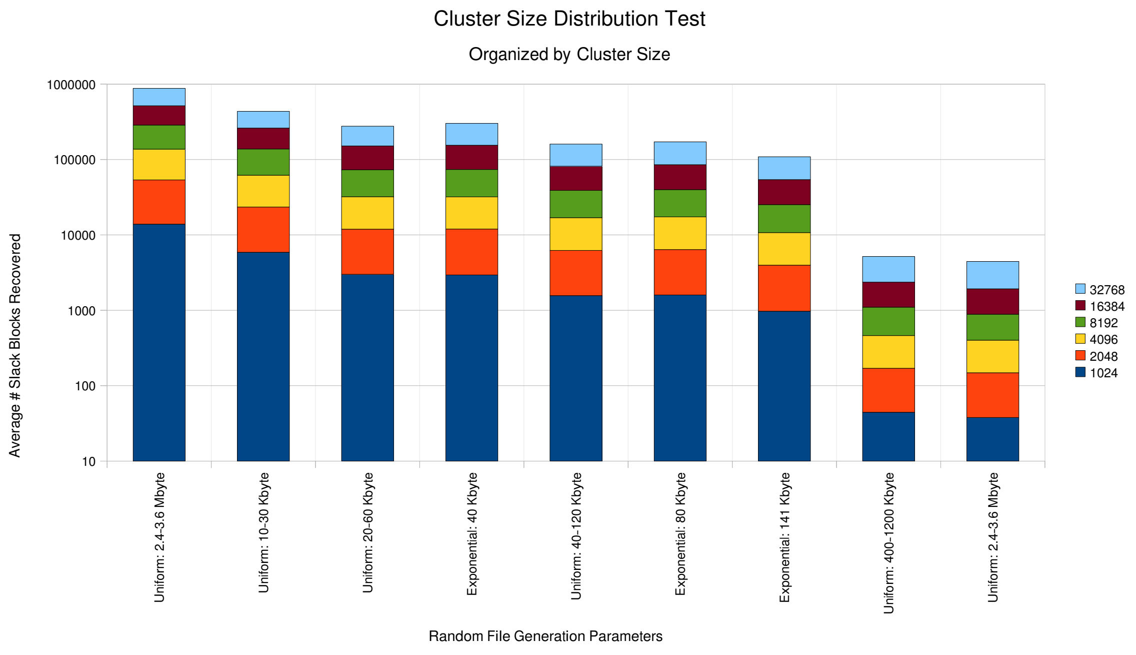

Figure 4.4: Results of Cluster Size Distribution Test (cluster size v. rf param)

An additional representation of the data, here grouped by random file pattern.

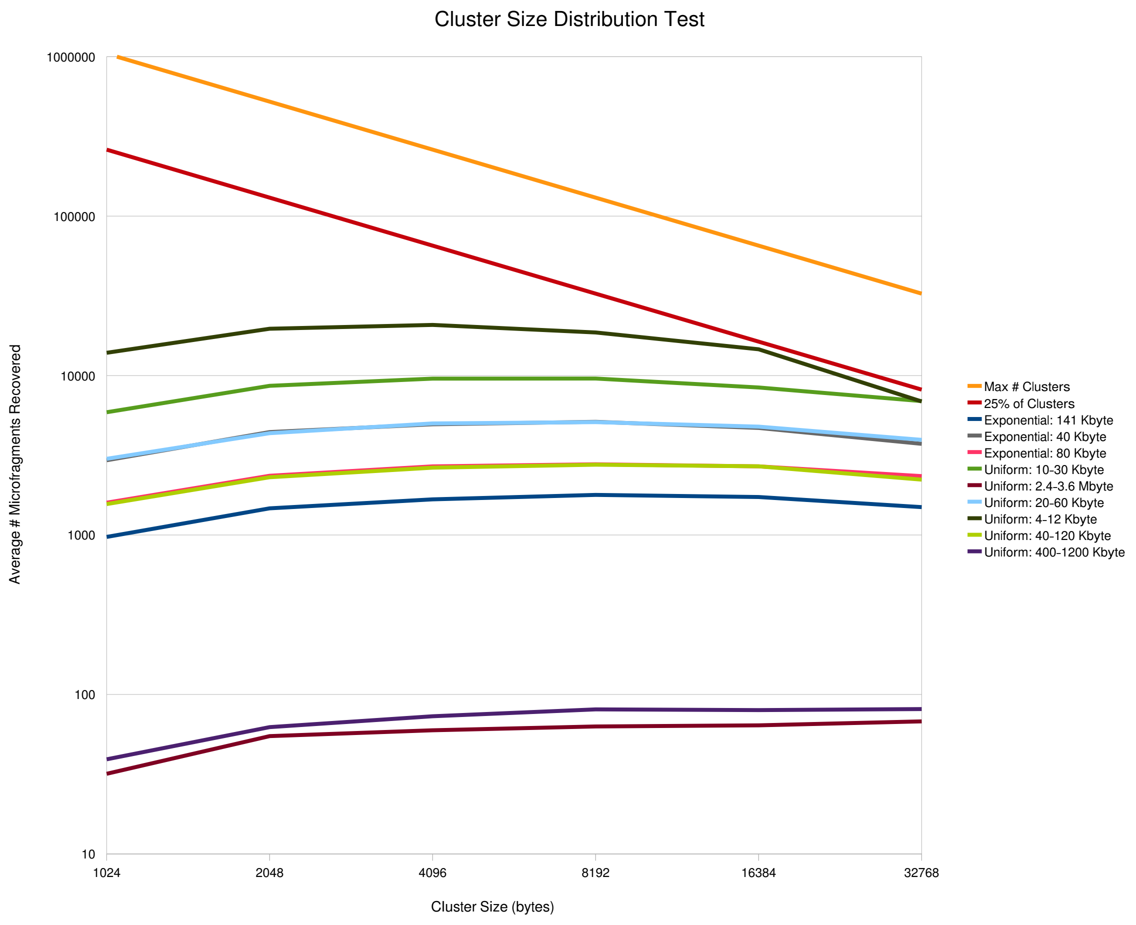

Figure 4.5: Results of Cluster Size Distribution Test (Microfragment Recovery)

Note: This chart is represented as a line graph for the ease of demonstrating trends. This data is not continuous.

4.2.3 '30 Repetitions' Test

As is seen from figure 4.7, the standard deviation very small for the smaller file sizes, but is significantly bigger for the larger files. There is also slightly more spread with our uniformly distributed random files than with our exponentially distributed random files.

This can be explained quite simply. A uniformly distributed file occupying between 10 and 30 kibibytes has an average length of 5.44 clusters and, barring tail alignment, an average of 3.54 blocks left in the tail and an 86% chance of having a microfragment. An exponentially distributed random file occupying approximately 20 kibibytes will have an approximate average length of 2.72 clusters, 3.83 slack blocks left, and a 91% chance of containing a microfragment, but these numbers vary. In Figure 4.6, we can see the variance between the probabilities for counts in block recovery. When this value is not zero, a microfragment is generated. The variance in the fragment generation percentages for our exponentially distributed random files accounts for the differences in precision.

With our results for larger file sizes, we see more of a spread because there are fewer files, and thus fewer microfragments generated. Larger files occupy more clusters per file, but, in the case of our uniform distribution, have a larger file size spread than its exponential equivalent. In the case of our larger tests, the cluster slack block availability frequencies approach a more uniform distribution.

The actual number of files used for overwriting the volume as well as specific sizes of these files were not collected with the automated tools.

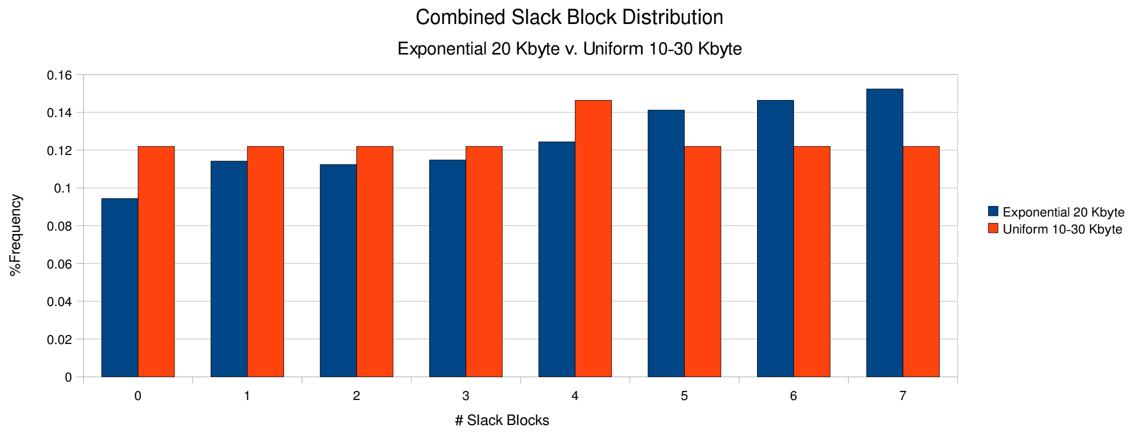

Figure 4.6: Uniform v. Exponentially distributed slack block probabilities

The distributions for the exponential data were taken from a sample of 5000 tests, and referenced in Table A.2.

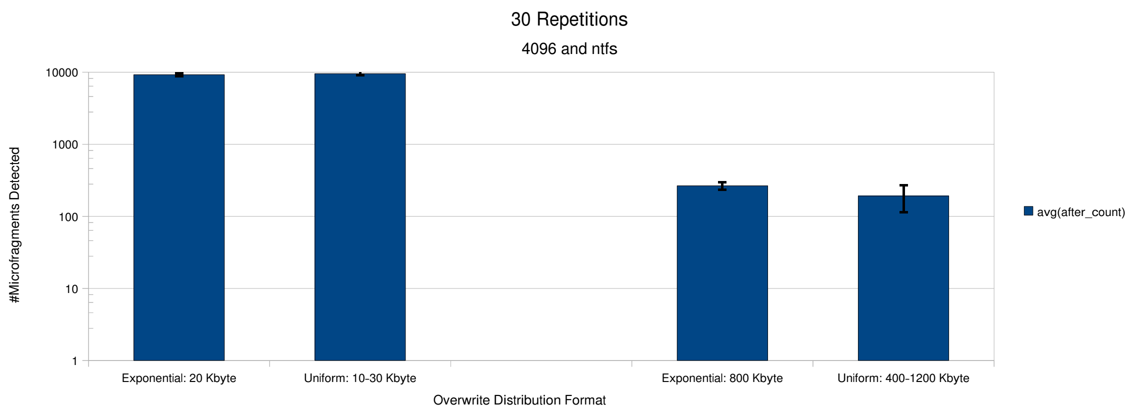

Figure 4.7: Observed Results of Repetitions 4096 NTFS

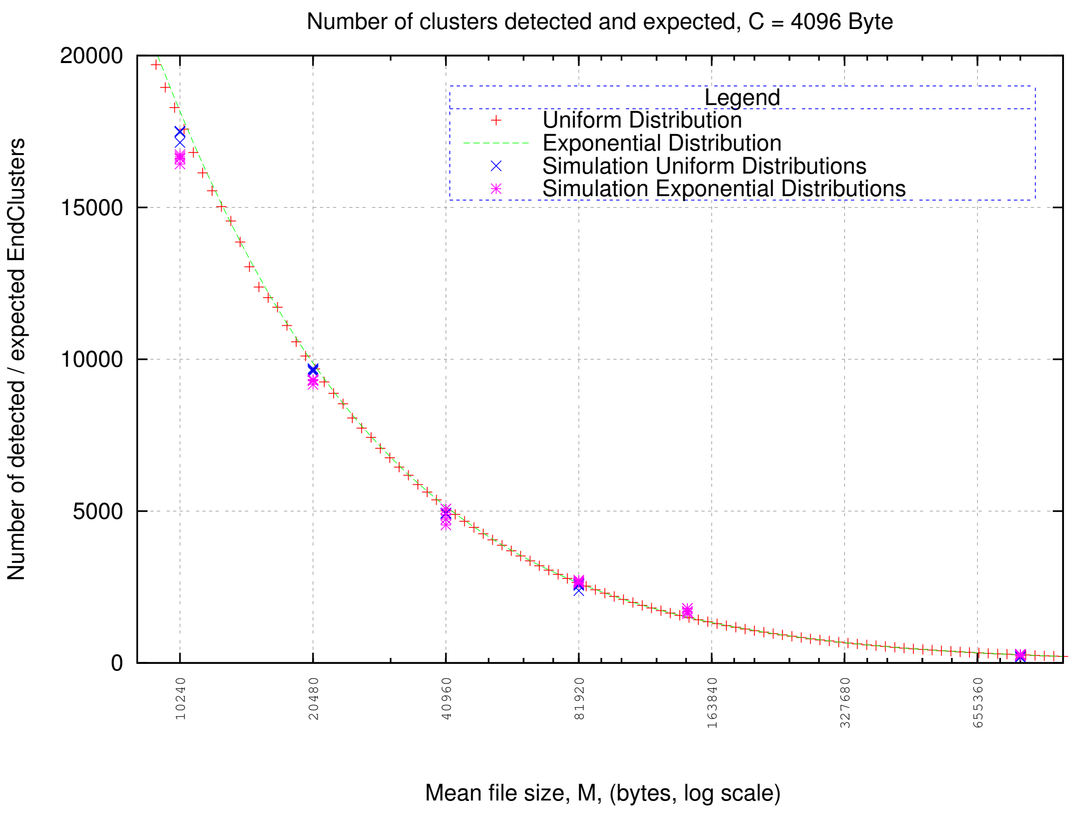

Figure 4.8: Recovered Microfragments (Experimental v. Analytical)

The general trend in the results (figure 4.8) versus our experimental model is quite evident. This is strong evidence that the models are accurate portrayals

4.2.4 'Rolling Hash' Tests

These tests yielded some interesting results in comparison with section 4.2.1. In figure 4.9 we can see a comparison between the number of microfragments recovered and the number of hashes matched by the rolling hash algorithm.

As can be observed, the recovery trends are quite similar. Surprisingly there is a difference by a factor of approximately 6 between the number of hashes matched and the number of microfragments detected by this series of tests. This could potentially demonstrate the frequency and occurrence of unallocated sectors in addition to the microfragments present.

.

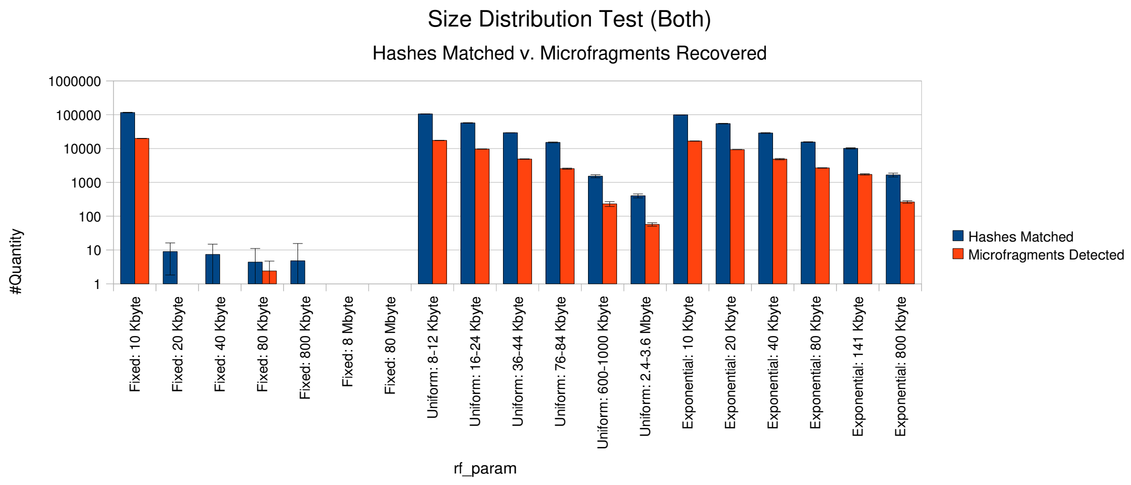

Figure 4.9: Size Distribution Test Comparison (Hashes v. Fragments)

The cluster size distribution set of rolling hash tests also demontrates an interesting pattern. With the exception of the 1024 byte cluster sizes, Figure 4.10 demonstrates a clear trend of hash recovery with respect to cluster size and random file parameter.

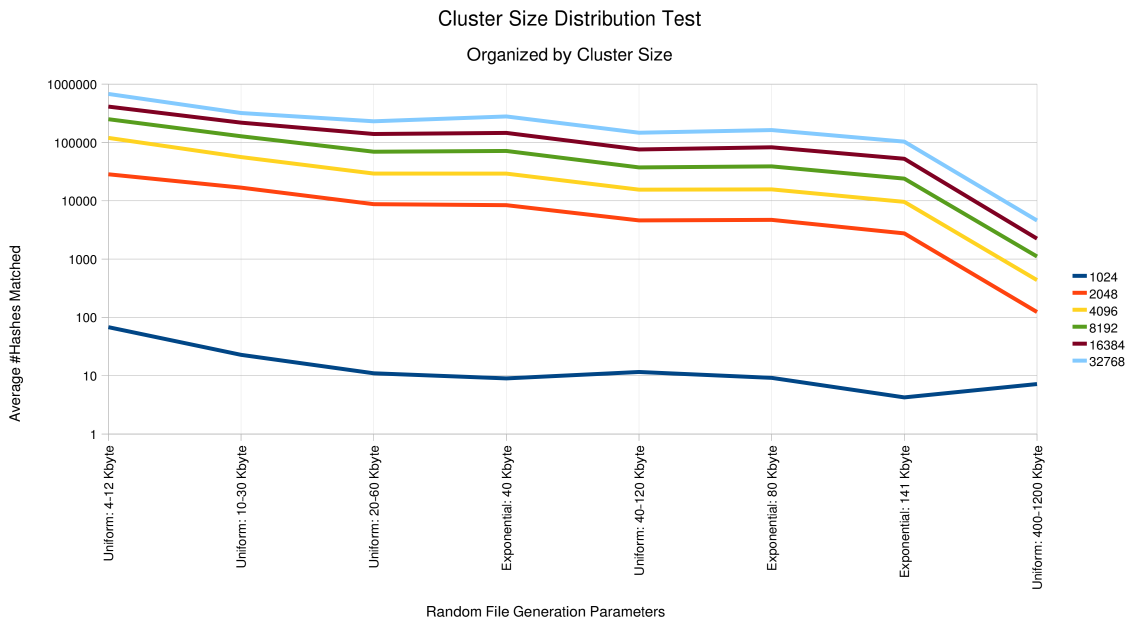

Figure 4.10: Size Distribution Tests (Hashes by Cluster Size)

When using a cluster size of 1024 bytes, each microfragment can only contain one block. Because of this, the microfragment recovery on a volume with this format parameter will not yield similar levels of data in comparison with a rolling hash analysis test.

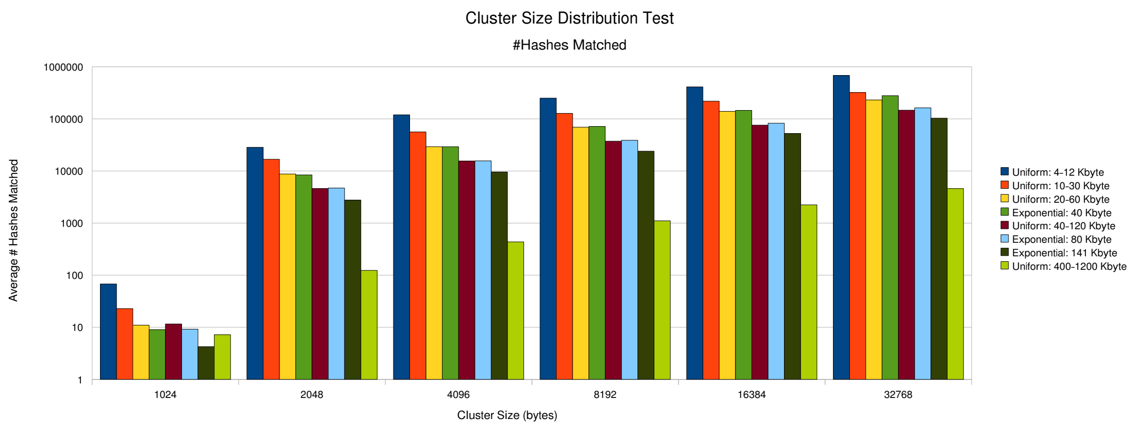

Figure 4.11: Cluster Size Tests (Rolling Hash)

The results for our cluster size distribution test seem to follow a similar trend. The following graph (figure 4.11) demonstrates only the results from our rolling hash test, as we would otherwise have too much data.

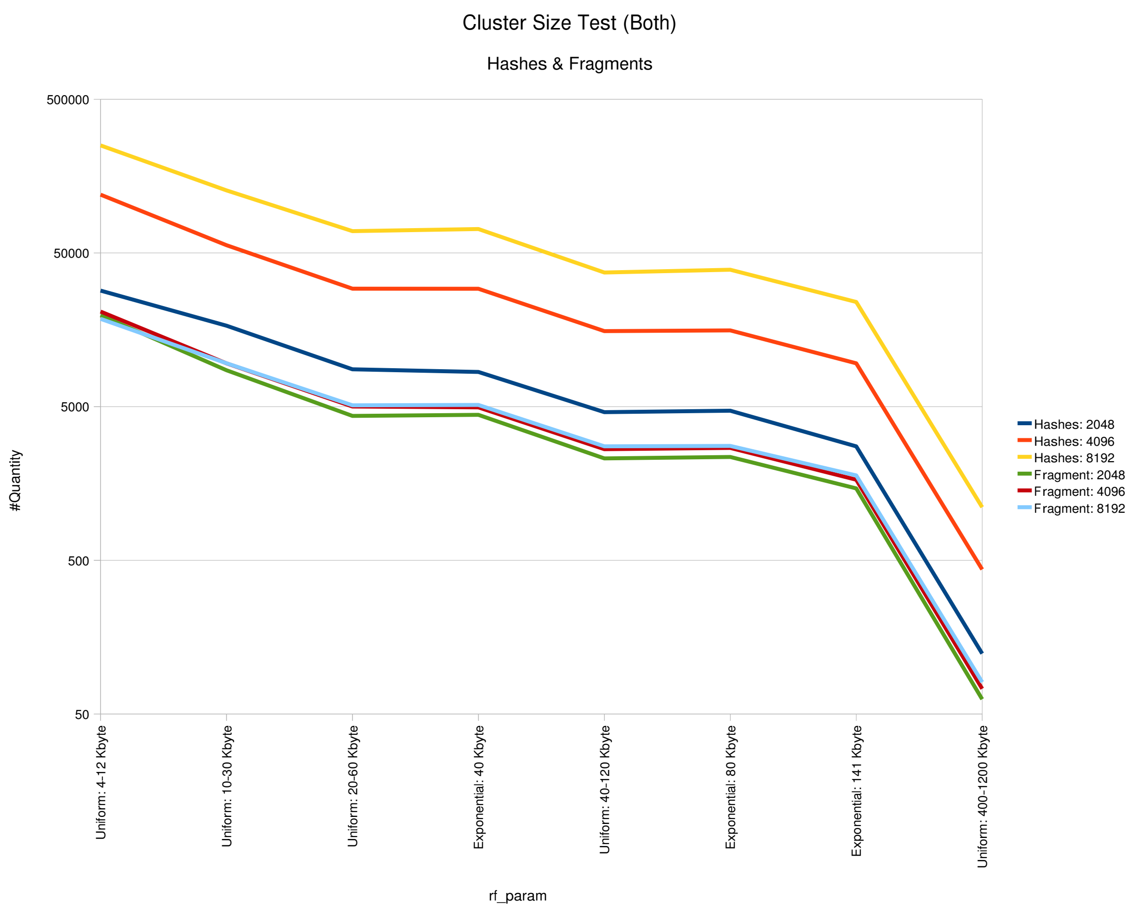

Figure 4.12: Cluster Size Tests (Ratio Demonstration)

Interesting to note, however, is that the multiplying factor between microfragment recovery and hash match is more dependent on cluster size, than of the overwritten data from the random files. Figure 4.12 uses a subset of the data from this test to demonstrate the independence, and figure 4.13 demonstrates the general trend in ratio between microfragment collection and hash block recovery.

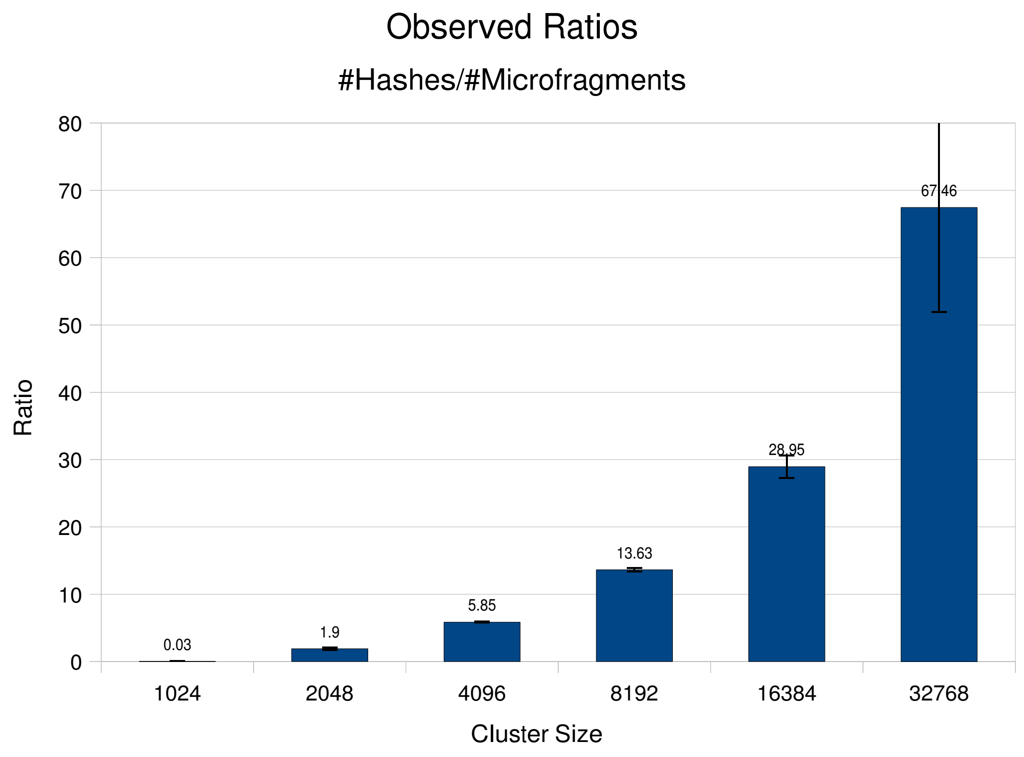

Figure 4.13: Microfragment v. Hash Ratios

4.3 Summary

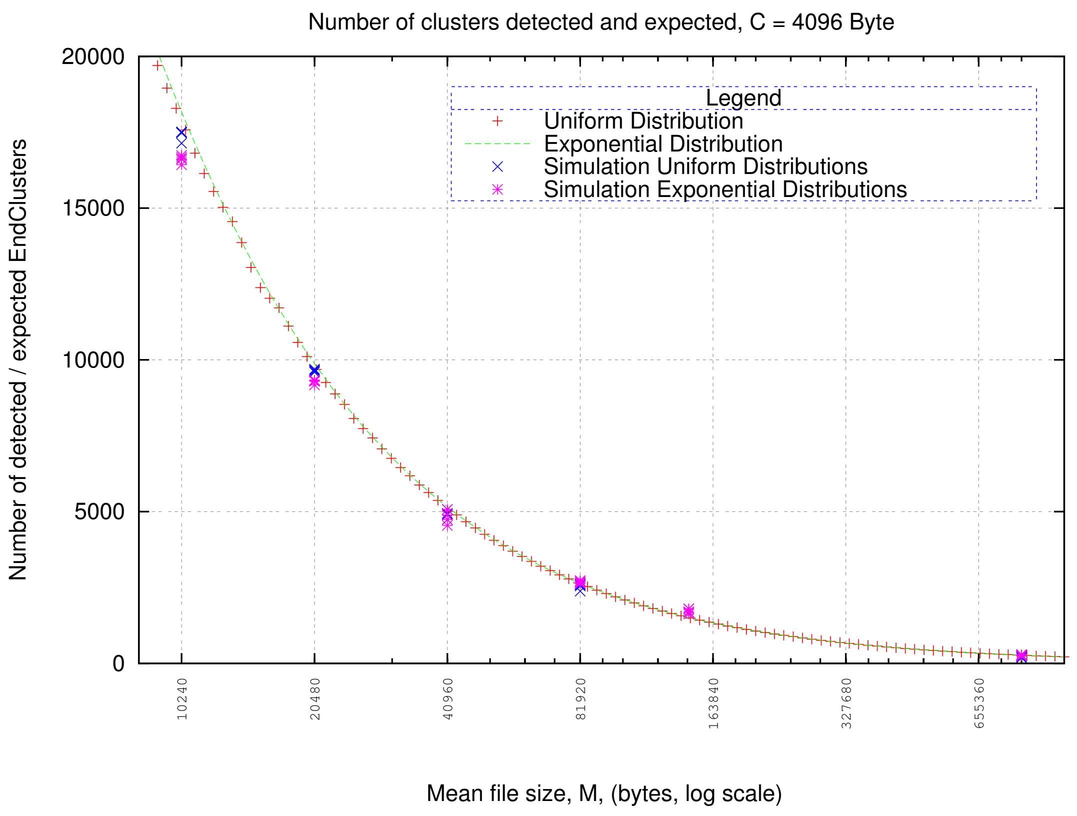

Figure 4.14: Block Recovery Trend Comparison

In this chapter the measured results are graphed and compared to our analytical model. There is a good agreement between these figures as shown in figure 4.14.

Chapter 5

Conclusion

5.1 Microfragment Collection

The analysis of the results clearly demonstrates that distribution of the overwriting files as well as filesystem format parameters have a direct and measurable effect upon the ability to recover file microfragments.

There was a strong quantitative agreement when comparing the measured results against expected results (figure 4.14), but further work could be performed to determine the number of hashes matched within deallocated versus tail sectors.

In conclusion, we see that cluster slack analysis presents an accurate and viable means with which we can recover file fragments for the purposes of digital forensics. In comparison with rolling hash analysis, we have a similar rate of recovery, but we have reduced the amount of time, data, and false positive rates we would normally see.

5.2 Future Work

Newer disk technologies relying on flash storage often have internal wear leveling mechanisms to increase the lifespan of the device. The Copy-on-Write technologies they employ may provide additional sources of duplicate hashes and more file fragments.

In addition, magnetic media storage densities are increasing rapidly, leading to a huge growth in available storage space. At present, there is a push by hard drive manufacturers to move to a standard of 4096 byte blocks at the hardware level[12]. From the standpoint of the operating system, this will not appear any differently but this may affect the number of microfragments recovered when new data is written.

Modeling and comparing the results of higher-order distributions (such as pareto) could also be useful as an indicator for expected recovery on an actual consumer device.

It could also be interesting to determine which factors affect the ratio of hash matches to microfragment recovery.

References

[1] Thijs Holleboom & Johan Garcia. Fragment Retention Characteristics in Slack Space - Analysis and Measurements. *Proceedings 2nd International Workshop on Security and Communucation Networks*, 2010.

[2] The FIVES initiative. http://fives.kau.se/.

[3] The Safer Internet Program. http://www.saferinternet.org/.

[4] Kilobyte. http://en.wikipedia.org/wiki/Kilobyte.

[5] Andries Brouwer. Properties of partition tables.

[6] Microsoft Corporation, http://technet.microsoft.com/en-us/library/cc781134 *How NTFS Works: Local File Systems.*

[7] Paragon Software Group. NTFS features. Technical report, http://www.paragon-software.com/ntfs/, Retrieved April 14th, 2010.

[8] NTFS MFT technical information. Technical report, http://www.ntfs.com/ntfs-mft.htm.

[9] The NT filesystem. Technical report, http://www.mcmillan.cx/ntfs.html.

[10] Technical report, The PC Guide (http://www.PCGuide.com), Site Version: 2.2.0 - Version Date: April 17, 2001.

[11] Johan Garcia & Thijs Holleboom. Retention of Micro-fragments in Cluster Slack - a first model. *Proceedings of IEEE Workshop on Information Forensics and Security*, 2009.

[12] Western Digitals Advanced Format: The 4k Sector Transition Begins. Technical report. http://www.anandtech.com/show/2888.

Appendix A

A.1 Graph Data

A.1.1 Uniform Distribution Calculations

#Blocks Remaining	Count	Probability
0	472	0.09
1	571	0.11
2	562	0.11
3	574	0.11
4	622	0.12
5	706	0.14
6	732	0.15
7	762	0.15

Table A.1: Graph Data for 'Uniform Block Distribution' Chart

This table provides the data used in figure 4.6. The data was generated by taking 5000 samples of 20 KiB file with an exponential differentiation.

A.1.2 'File Size Distribution' Test

	Microfragment Analysis		Rolling Hash Matches	
rf_param	Avg #Microfragments	Microfragment stdev	Avg #Hashes Matched	Stdev
Fixed: 10 Kbyte	19917.2	35.68	116055	2406.86
Fixed: 20 Kbyte	0.2	0.45	9	7.18
Fixed: 40 Kbyte	0.8	0.45	7.4	7.5
Fixed: 8 Mb	0.6	0.55	0	0
Fixed: 80 Kbyte	2.4	2.3	4.4	6.66
Fixed: 80 MB	0.2	0.45	0	0
Fixed: 800 Kbyte	0	0	4.8	10.73
Uniform: 16-24 Kbyte	9643	38.08	57213.2	1153.49
Uniform: 2.4-3.6 Mbyte	57.2	6.53	29301.2	247.92
Uniform: 36-44 Kbyte	4882.2	82.15	1526.8	154.26
Uniform: 600-1000 Kbyte	231.2	37.63	15197.6	421.74
Uniform: 76-84 Kbyte	2547.6	97.68	105619.33	46.01
Uniform: 8-12 Kbyte	17425.6	161.26	401.2	53.12
Exponential: 10 Kbyte	16616.8	117.63	99141.25	225.59
Exponential: 141 Kbyte	1718	76.84	10112.8	421.66
Exponential: 20 Kbyte	9288	66.32	54604.5	964.87
Exponential: 40 Kbyte	4862.6	226.58	28799.4	822.99
Exponential: 80 Kbyte	2671.6	41.69	15591.4	350.56
Exponential: 800 Kbyte	263	22.28	1662	215.25

Table A.2: Graph Data for 'File Size Distribution' Test

This table provides the data used in figures 4.1, 4.9, and 4.11

A.1.3 'Cluster Size Distribution' Test

Cluster Size	rf_param	Avg #Microfragments	Microfragment stdev
1024	Exponential: 141 Kbyte	972.4	33.63
2048	Exponential: 141 Kbyte	1469.4	29.61

Continued on next page

Cluster Size	rf_param	Avg #Microfragments	Microfragment stdev
4096	Exponential: 141 Kbyte	1673.6	55.44
8192	Exponential: 141 Kbyte	1785	49.17
16384	Exponential: 141 Kbyte	1731.2	41.08
32768	Exponential: 141 Kbyte	1495.6	23.22
1024	Exponential: 40 Kbyte	2942.6	61.69
2048	Exponential: 40 Kbyte	4423	67.83
4096	Exponential: 40 Kbyte	4948	24.24
8192	Exponential: 40 Kbyte	5127.6	72.78
16384	Exponential: 40 Kbyte	4694	88.88
32768	Exponential: 40 Kbyte	3731	38.31
1024	Exponential: 80 Kbyte	1595.4	23.52
2048	Exponential: 80 Kbyte	2353	53.51
4096	Exponential: 80 Kbyte	2694.6	52.84
8192	Exponential: 80 Kbyte	2776.6	55.4
16384	Exponential: 80 Kbyte	2695.2	50.23
32768	Exponential: 80 Kbyte	2340	43.12
1024	Uniform: 10-30 Kbyte	5888.4	45.59
2048	Uniform: 10-30 Kbyte	8623.2	54.64
4096	Uniform: 10-30 Kbyte	9564.8	38.21
8192	Uniform: 10-30 Kbyte	9579	21.33
16384	Uniform: 10-30 Kbyte	8419	28.37
32768	Uniform: 10-30 Kbyte	6916	10.56
1024	Uniform: 2.4-3.6 Mbyte	31.8	8.01
2048	Uniform: 2.4-3.6 Mbyte	54.8	5.07
4096	Uniform: 2.4-3.6 Mbyte	59.6	8.96
8192	Uniform: 2.4-3.6 Mbyte	63	6
16384	Uniform: 2.4-3.6 Mbyte	64	5.34
32768	Uniform: 2.4-3.6 Mbyte	67.8	7.92
1024	Uniform: 20-60 Kbyte	3005.4	46.8
2048	Uniform: 20-60 Kbyte	4350.8	73.85

Continued on next page

Cluster Size	rf_param	Avg #Microfragments	Microfragment stdev
4096	Uniform: 20-60 Kbyte	5009	13
8192	Uniform: 20-60 Kbyte	5102.6	41.45
16384	Uniform: 20-60 Kbyte	4782.2	27.8
32768	Uniform: 20-60 Kbyte	3952	24.96
1024	Uniform: 4-12 Kbyte	13897.4	43.67
2048	Uniform: 4-12 Kbyte	19656.4	96.7
4096	Uniform: 4-12 Kbyte	20783.25	143.26
8192	Uniform: 4-12 Kbyte	18627.8	41.22
16384	Uniform: 4-12 Kbyte	14615.6	21.7
32768	Uniform: 4-12 Kbyte	6878.6	41.45
1024	Uniform: 40-120 Kbyte	1563.6	37.82
2048	Uniform: 40-120 Kbyte	2302.8	48.63
4096	Uniform: 40-120 Kbyte	2643.8	27.64
8192	Uniform: 40-120 Kbyte	2759.6	21.13
16384	Uniform: 40-120 Kbyte	2696	21.93
32768	Uniform: 40-120 Kbyte	2225	19.43
1024	Uniform: 400-1200 Kbyte	39.2	4.76
2048	Uniform: 400-1200 Kbyte	62.4	4.72
4096	Uniform: 400-1200 Kbyte	73	5.2
8192	Uniform: 400-1200 Kbyte	80.6	3.36
16384	Uniform: 400-1200 Kbyte	79.8	6.06
32768	Uniform: 400-1200 Kbyte	81	7.11

Table A.3: Graph Data for 'Cluster Size Distribution' Test

This table provides the data used in figures 4.3, 4.4, and 4.5

A.1.4 '30 Repetitions' Test

rf_param	Avg #Microfragments	Microfragment stdev
Exponential: 20 Kbyte	9229.63	229.35
Uniform: 10-30 Kbyte	9545.48	246.88
Exponential: 800 Kbyte	265.31	16.47
Uniform: 400-1200 Kbyte	191.97	38.71

Table A.4: Graph Data for '30 Repetitions' Test

This table provides the data used in figure 4.7.

A.1.5 'Rolling Hash' Tests

cluster size	rf param	Hashes Matched	stdev	compchunk	stdev	Ratio
1024	Uniform: 4-12 Kbyte	68	8.76	13897.4	43.67	0
1024	Uniform: 10-30 Kbyte	22.8	2.77	5888.4	45.59	0
1024	Uniform: 20-60 Kbyte	11	2.16	3005.4	46.8	0
1024	Exponential: 40 Kbyte	9	3.32	2942.6	61.69	0
1024	Uniform: 40-120 Kbyte	11.6	2.41	1563.6	37.82	0.01
1024	Exponential: 80 Kbyte	9.2	3.19	1595.4	23.52	0.01
1024	Exponential: 141 Kbyte	4.25	1.5	972.4	33.63	0
1024	Uniform: 400-1200 Kbyte	7.2	8.81	39.2	4.76	0.18
2048	Uniform: 4-12 Kbyte	28454.5	18971.03	19656.4	96.7	1.45
2048	Uniform: 10-30 Kbyte	16818.2	495.38	8623.2	54.64	1.95
2048	Uniform: 20-60 Kbyte	8751.2	255.67	4350.8	73.85	2.01
2048	Exponential: 40 Kbyte	8410.8	217.67	4423	67.83	1.9
2048	Uniform: 40-120 Kbyte	4603.6	247.09	2302.8	48.63	2
2048	Exponential: 80 Kbyte	4703.8	132.19	2353	53.51	2
2048	Exponential: 141 Kbyte	2757.6	125.72	1469.4	29.61	1.88
2048	Uniform: 400-1200 Kbyte	123.6	15.95	62.4	4.72	1.98
4096	Uniform: 4-12 Kbyte	119780.25	391.25	20783.25	143.26	5.76
4096	Uniform: 10-30 Kbyte	56146	1139.02	9564.8	38.21	5.87

Continued on next page

cluster size	rf param	Hashes Matched	stdev	compchunk	stdev	Ratio
4096	Uniform: 20-60 Kbyte	29276	643.19	5009	13	5.84
4096	Exponential: 40 Kbyte	29242.2	830.91	4948	24.24	5.91
4096	Uniform: 40-120 Kbyte	15506	273.52	2643.8	27.64	5.87
4096	Exponential: 80 Kbyte	15674.8	204.87	2694.6	52.84	5.82
4096	Exponential: 141 Kbyte	9573.6	351.45	1673.6	55.44	5.72
4096	Uniform: 400-1200 Kbyte	436.2	42.27	73	5.2	5.98
8192	Uniform: 4-12 Kbyte	250555.6	941.58	18627.8	41.22	13.45
8192	Uniform: 10-30 Kbyte	127684.67	1387.36	9579	21.33	13.33
8192	Uniform: 20-60 Kbyte	69324.2	636.17	5102.6	41.45	13.59
8192	Exponential: 40 Kbyte	71565.5	1629.26	5127.6	72.78	13.96
8192	Uniform: 40-120 Kbyte	37314	963.81	2759.6	21.13	13.52
8192	Exponential: 80 Kbyte	38890.33	613.02	2776.6	55.4	14.01
8192	Exponential: 141 Kbyte	23989.33	1251.15	1785	49.17	13.44
8192	Uniform: 400-1200 Kbyte	1107.4	88.02	80.6	3.36	13.74
16384	Uniform: 4-12 Kbyte	411896.2	2254.81	14615.6	21.7	28.18
16384	Uniform: 10-30 Kbyte	218746.4	1581.79	8419	28.37	25.98
16384	Uniform: 20-60 Kbyte	139592.2	1507.24	4782.2	27.8	29.19
16384	Exponential: 40 Kbyte	145732.2	1750.46	4694	88.88	31.05
16384	Uniform: 40-120 Kbyte	75823.8	1997.98	2696	21.93	28.12
16384	Exponential: 80 Kbyte	82618	817.43	2695.2	50.23	30.65
16384	Exponential: 141 Kbyte	52462.6	3732.51	1731.2	41.08	30.3
16384	Uniform: 400-1200 Kbyte	2242.4	119	79.8	6.06	28.1
32768	Uniform: 4-12 Kbyte	678958	2408.65	6878.6	41.45	98.71
32768	Uniform: 10-30 Kbyte	319959.4	2628.59	6916	10.56	46.26
32768	Uniform: 20-60 Kbyte	230784.8	1386.56	3952	24.96	58.4
32768	Exponential: 40 Kbyte	279487.8	7654.1	3731	38.31	74.91
32768	Uniform: 40-120 Kbyte	146799	2452.7	2225	19.43	65.98
32768	Exponential: 80 Kbyte	163100.6	4234.95	2340	43.12	69.7
32768	Exponential: 141 Kbyte	103249.8	2058.69	1495.6	23.22	69.04
32768	Uniform: 400-1200 Kbyte	4592	467.59	81	7.11	56.69

This table provides the data used in figures 4.10, 4.11, 4.12, and 4.13.

A.2 Fragment Analysis

A.2.1 findGenFrag.c

```
/*

Adapted from code taken from Johan Garcia. - Zak Blacher 2010

NOTE:  When compiling use  gcc -Wall -D_FILE_OFFSET_BITS=64 -o findGenFrag findGenFrag.c
to make sure that the file handling functions in glibs will be able to handle
files>2Gb. See also http://www.suse.de/~aj/linux_lfs.html

*/

//----- Include files -------------------------------------------------------
#include <stdio.h>                // Needed for printf() and feof()
#include <stdlib.h>               // Needed for exit(), atof(), and qsort()
#include <string.h>               // Needed for strcmp()

//----- Defines -------------------------------------------------------------

#define READ_BLOCKSIZE      10*1024*1024    //Should be multiple of chunksize
#define READ_MARGIN         64*1024
#define PRINTINTERVAL       1

//----- Globals -------------------------------------------------------------
//----- Type defines ----------------------------------------------------------
typedef unsigned char      byte;   // Byte is a char
typedef unsigned short int word16; // 16-bit word is a short int
typedef unsigned int       word32; // 32-bit word is an int

//----- Constant defines ------------------------------------------------------
#define      FALSE            0   // Boolean false
#define       TRUE            1   // Boolean true

//----- Global variables ------------------------------------------------------
int64_t   N=0;                     // Number of bytes in checkfile

//===========================================================================
//=  Main program                                                           =
//===========================================================================


void print_usage(char *name)
{
 printf  ("Usage: %s \n"
   " [-d <debug>] Debug level (0-2)\n"
     " [-m] Machine readable output (cannot combine with -d)\n"
   " [-c <checkfile>] file to check for matches\n",name);
exit (1);
}


int main(int argc, char *argv[])
{
 FILE   *checkfileptr=0;                    // File pointer to input file #2
 word32 debug=0l,machine_readable=0l;
 char *checkfile=NULL;
 word32 i,j;                        // Loop counter
 char *buffer;
 int printinterval = PRINTINTERVAL;
 int argctr=0;
 word32 read=0l,nr_startheaders=0l,nr_endheaders=0l,nr_fragments=0l;
 int64_t  slackdata=0ll;
 unsigned int filenumber,chunknr,chunksize;


 if (argc < 2 ) {
   print_usage(argv[0]);
 }

 while (++argctr < argc) {
   if ( (*(argv[argctr])) != '-' )  print_usage (argv[0]);
   switch (*(argv[argctr]+1)) {
     case 'd' : debug=atoi(argv[++argctr]); break;
     case 'm' : machine_readable=1; break;
     case 'c' : checkfile=argv[++argctr]; break;

   }
   if (debug && machine_readable)
     exit (1);
 }

 buffer=calloc(READ_BLOCKSIZE,sizeof(byte));

 if (checkfile != NULL) {
   checkfileptr=fopen(checkfile,"r");
   if (checkfileptr==NULL) {fprintf(stderr,"Could not open file %s",checkfile); exit (1);}
 }

 N=0ll;
 read=0;
  do {
       read=fread(buffer,sizeof(char),READ_BLOCKSIZE,checkfileptr);
N += read;
if (!(--printinterval) && !machine_readable) {
  fprintf(stderr,"Searched: %lld\r",N);
  fflush (stdout);
  printinterval=PRINTINTERVAL;
}
for (i=0;i < read ;i++) {
  if (buffer[i]=='F') { //
    if (!strncmp(buffer+i,"FIVB",4)) {
      nr_startheaders++;
      if (debug>1) printf ("\nStartheader found at position %lld (non-fp)   ",N-read+i);
      sscanf(buffer+i+23,"%d",&filenumber);  // Evil magic nrs correlated to output format...
      sscanf(buffer+i+39,"%d",&chunknr);
      sscanf(buffer+i+55,"%d",&chunksize);
      if (debug>2) printf ("Filenumber: %9u, Chunknumber: %9u (Chunksize:%5u)",
   filenumber,chunknr,chunksize);
      if (!strncmp(buffer+i+chunksize-4,"FIVE",4)) {  //Complete chunk?
nr_endheaders++;
slackdata += chunksize;
      } else {
for (j=0;buffer[i+64+j]=='J';j++);
slackdata += j;
      }
      if (strncmp(buffer+i+chunksize,"FIVB",4)) {  //If not followed by chunk, increase fragcounter
nr_fragments++;
      }
    }
  }
}
  }    while (read == READ_BLOCKSIZE);

  fclose(checkfileptr);
  if (machine_readable) {
    fprintf(stdout,"%u;%lld;%lld;%u;%u\n",nr_fragments,N,slackdata,nr_endheaders,nr_endheaders-nr_startheaders);
    fprintf(stdout,"microfragments: %u\n", nr_fragments);
    fprintf(stdout,"processed-bytes: %lld\n", N);
    fprintf(stdout,"slack-bytes: %lld\n", slackdata);
    fprintf(stdout,"complete-chunks: %u\n", nr_endheaders);
    fprintf(stdout,"incomplete-chunks: %u\n", nr_endheaders-nr_startheaders);

  } else {
    fprintf(stdout,"\nFinished.\n");
    fprintf(stdout,"%lld bytes processed",N);
    fprintf(stdout,"%u microfragments found\n",nr_fragments);
    fprintf(stdout,"%lld bytes of slack data found\n",slackdata);
    fprintf(stdout,"%u complete chunks found\n",nr_endheaders);
    fprintf(stdout,"%u incomplete chunks found\n",nr_endheaders-nr_startheaders);
    fprintf(stdout,"%u last chunksize detected\n",chunksize);

    // Output closing trailer
    printf("------------------------------------------------------- \n");
  }
 return 0;

}
```

A.2.2 genDistrFile.c

```
/*

Adapted from code taken from Johan Garcia. - Zak Blacher 2010

TODO:
* Think about when to use llong and when to use double
* Think about if  the random generating function is sufficient,
 It uses only ints to store seed
Compile with: gcc -Wall -std=gnu99 -lm genDistrFile.c
*/


//----- Include files -------------------------------------------------------
#include <stdio.h>                 // Needed for printf() and feof()
#include <stdlib.h>                // Needed for exit(), atof(), and qsort()
#include <string.h>                // Needed for strcmp()
#include <limits.h>                // Needed for LLONG_MAX etc
#include <errno.h>                 // Needed for errno ...
#include <sys/types.h>             // Needed for stat()
#include <sys/stat.h>              // Needed for stat()
#include <sys/time.h>              // Needed for gettimeofday()
#include <time.h>                  // Needed for stat()
#include <math.h>                  // Needed for pow(), sqrt(), log(), ..


//----- Defines --------- -------------------------------------------------------

#define  BUFFERSIZE  (16*1024*1024)  //Size of writebuffer for efficient writing

//----- Global variables -------------------------------------------------------

int debug=0;               //Amount of debug info to output
enum DISTROS {STATIC, UNIFORM, NORMAL, TRUNC_EXPON, EXPON, PARETO, BOUNDED_PARETO, FILE_SEQ, FILE_RAND, END_MARKERD} distrib
enum AMOUNTS {NRFILES, MIN_SIZE, MAX_SIZE, EXACT_SIZE, END_MARKERA} amounttype = END_MARKERA;
enum FILLTYPE{CHUNK, RANDOM, TEXT, END_MARKERF} filltype = CHUNK;
unsigned int chunksize=512;                //Size of chunks with identificators for generated files
unsigned int filenumber=0;
char *fillstring;
```

```
//----- Function prototypes ----------------------------------------------------


char *write_file (char *outputdir, long long filesize);
double pareto(double a, double k);
double bpareto(double a, double k, double p);
double expon(double x);
double norm(double mean, double std_dev);
double rand_val(int seed);
double unif(double min, double max);
void checkdir (char *outputdir);
//=============================================================================
//=  Main program                                                             =
//=============================================================================
/*
Filerna som genereras skall ha fljande karakteristik:

Bestr av ett antal chunkar, varje chunk har:

Startheader   --  Paddad med 119 -- slutheader

Startheader: FIVB, 4bytes filnummer, 4bytes chunknummer, 4bytes chunkstorlek
Slutheader FIVE, 4bytes filnummer, 4bytes chunknummer, 4bytes chunkstorlek

*/


long long getLlong(char *valstring)
{
 char *endptr;
 long long val;


 errno = 0;    /* To distinguish success/failure after call */
 val = strtoll(valstring, &endptr, 10); //Use base 10

 /* Check for various possible errors */

 if ((errno == ERANGE && (val == LLONG_MAX || val == LLONG_MIN))
     || (errno != 0 && val == 0)) {
   perror("strtoll");
   exit(EXIT_FAILURE);
 }

 if (endptr == valstring) {
   fprintf(stderr, "No digits were found\n");
   exit(EXIT_FAILURE);
 }

 /* If we got here, strtol() successfully parsed a number */


 if (*endptr != '\0') {        /* Not necessarily an error... */
   if (!strncmp (endptr,"KbKi)  val=val*1024;
   else if (!strncmp (endptr,"Mb",2))  val=val*1024*1024;
   else if (!strncmp (endptr,"Gb",2))  val=val*1024*1024*1024;
   else if (!strncmp (endptr," ",1))  ; // For size file with more text on line
     else {
printf("Further characters after number: %s\n", endptr);
exit(EXIT_FAILURE);
     }
 }

 if (debug>0)  printf("getLlong returns %lld\n", val);
 return val;
```

```
}


void print_usage(char *name)
{
 printf  ("Usage: %s\n"
   " Distribution characteristics for generated files, select one: \n"
   " [-s <static size>]                      One static file size\n"
   " [-u <lower limit> <upper limit> ]      Uniformly distributed values. Limits are inclusive. \n"
   " [-n <mean> <stddev>]                    Normal\n"
   " [-t <mean> <shape> <max>]               Truncated exponential\n"
   " [-e <mean> <shape>]                     Exponential\n"
   " [-p <mean> <shape>]                     Pareto\n"
   " [-b <mean> <shape>]                     Bounded pareto\n"
   " [-f <input filename>]                   Read sizes from file sequentially\n"
   " [-g <input filename>]                   Random draw from file with sizes\n"
   " Output generation options, select one or both: \n"
   " [-o <output file directory>]            Directory to store the generated files in        \n"
   " [-w <filename>]                         File to log the file sizes in \n"
   " Output amount options, select one: \n"
   " [-c <number of files>]                  The number of files generated\n"
   " [-k <min total file size>]              Minimum amount of cumulative file sizes, total may be larger. \n"
   " [-l <max total file size>]              Maximum  amount of cumulative file sizes, total may be less. \n"
   " [-m <exact total file size>]            Exact amount of cumulative file sizes. Last file may be truncated. \n"
   " Other options: \n"
   " [-z <chunksize>]                        Chunksize to be used for contents in files. Default 512\n"
   " [-q ]                                   Fill genereated files with random bytes,not chunks\n"
   " [-y <text_string>]                      Fill with specified string. Spaces not allowed\n"
   " [-r <seed value>]                       Value to use for random seed\n"
   " [-h]                                    Show this help\n"
          " [-d <debug>]                              Print debug messages according to debug level (0-2)\n"
   "\n"
   " Byte is default for all sizes, use KbKiGb for KbKiMbyte. example: 500KbKi
   ,name);
exit (EXIT_FAILURE);
}


int main(int argc, char *argv[])
{
 FILE   *srcfileptr=0;                       // File pointer to file with sizes
 FILE   *printfileptr=0;                     // File pointer to size log file
 char *srcfile=NULL,*outputdir=NULL,*printfile=NULL;
 long long staticsize=0, startlimit=0, endlimit=0, meansize=0, stddev=0;
 double shape=0;
 long long numberoffiles=0, totalfilesize=0, size=0, generatedfiles=0, generatedsize=0;
 int argctr=0, seed=0;
 int sizelines = 0, currentline=0, pos=0;
 long long *filesizearray = NULL;


 if (argc < 2 ) {
   print_usage(argv[0]);
 }

 while (++argctr < argc) {
   if ( (*(argv[argctr])) != '-' )  {
     fprintf (stderr,"Parsing error at %s\n",argv[argctr]);
     print_usage (argv[0]);
   }
   switch (*(argv[argctr]+1)) {
     case 's' :
distrib = STATIC;
staticsize = getLlong(argv[++argctr]);
break;
```

```
     case 'u' :
distrib = UNIFORM;
startlimit=getLlong(argv[++argctr]);
endlimit=getLlong(argv[++argctr]);
break;
     case 'n' :
distrib = NORMAL  ;
meansize=getLlong(argv[++argctr]);
stddev = getLlong(argv[++argctr]);
break;
     case 't' :
distrib = TRUNC_EXPON;
meansize=getLlong(argv[++argctr]);
shape = strtod (argv[++argctr],NULL);
endlimit=getLlong(argv[++argctr]);
if (shape == 0) {
    fprintf (stderr,"Could not parse shape parameter %s\n",argv[argctr]);
    exit (EXIT_FAILURE);
}
endlimit=getLlong(argv[++argctr]);
break;
     case 'e' :
distrib = EXPON  ;
meansize=getLlong(argv[++argctr]);
shape = strtod (argv[++argctr],NULL);
if (shape == 0) {
    fprintf (stderr,"Could not parse shape parameter %s\n",argv[argctr]);
    exit (EXIT_FAILURE);
}
break;
     case 'p' :
distrib = PARETO  ;
meansize=getLlong(argv[++argctr]);
shape = strtod (argv[++argctr],NULL);
if (shape == 0) {
    fprintf (stderr,"Could not parse shape parameter %s\n",argv[argctr]);
    exit (EXIT_FAILURE);
}
break;
     case 'b' :
distrib = BOUNDED_PARETO  ;
meansize=getLlong(argv[++argctr]);
shape = strtod (argv[++argctr],NULL);
endlimit=getLlong(argv[++argctr]);
if (shape == 0) {
    fprintf (stderr,"Could not parse shape parameter %s\n",argv[argctr]);
    exit (EXIT_FAILURE);
}
break;
     case 'f' :
distrib = FILE_SEQ ;
srcfile=argv[++argctr];
break;
     case 'g' :
distrib = FILE_RAND ;
srcfile=argv[++argctr];
break;
     case 'o' :
outputdir=argv[++argctr];
break;
     case 'c' :
amounttype = NRFILES;
numberoffiles=getLlong(argv[++argctr]);
break;
     case 'k' :
```

```
amounttype = MIN_SIZE;
totalfilesize=getLlong(argv[++argctr]);
break;
     case 'l' :
amounttype = MAX_SIZE;
totalfilesize=getLlong(argv[++argctr]);
break;
     case 'm' :
amounttype = EXACT_SIZE;
totalfilesize=getLlong(argv[++argctr]);
break;
     case 'w' :
printfile=argv[++argctr];
break;
     case 'z' :
filltype = CHUNK;
       chunksize = (int)getLlong(argv[++argctr]);
       if ( chunksize == 0) {fprintf (stderr,"Chunk size 0 no allowed\n"); exit (EXIT_FAILURE); }
break;
     case 'q' :
filltype = RANDOM;
break;
     case 'y' :
filltype = TEXT;
fillstring = argv[++argctr];
break;
     case 'r' :
       seed = (int)getLlong(argv[++argctr]);
       if ( seed == 0) {fprintf (stderr,"Seed value 0 no allowed\n"); exit (EXIT_FAILURE); }
rand_val(seed);
break;
     case 'h' :
print_usage (argv[0]);
break;
     case 'd' :
debug = (int)getLlong(argv[++argctr]);
break;
   default:
print_usage (argv[0]);
break;
   }
 }

 if (!seed) {
   struct timeval tv;
   gettimeofday(&tv,NULL);
   rand_val((int)tv.tv_usec);
   if (debug) printf ("Timebased seed :%d",(int)tv.tv_usec);
 }

 if (numberoffiles != 0 && totalfilesize != 0) {
   fprintf(stderr,"Not possible to both specify number of files and total file size\n\n");
   exit (EXIT_FAILURE);
 }

 if (BUFFERSIZE%chunksize != 0) {
     fprintf(stderr,"Illegal chunksize (i.e.BUFFERSIZE mod chunksize !=0) \n (%d mod %d)\n",BUFFERSIZE,chunksize);
     exit (EXIT_FAILURE);
 }

 if (srcfile != NULL) {
   srcfileptr=fopen(srcfile,"r");
   if (srcfileptr==NULL) {fprintf(stderr,"Could not open file %s\n",srcfile); exit (EXIT_FAILURE);}

   char linebuf[5000];
   while (fgets(linebuf,5000-1,srcfileptr) != NULL)
     sizelines++;
   rewind(srcfileptr);
   filesizearray = malloc(sizelines*sizeof(long long));

   while (fgets(linebuf,5000-1,srcfileptr) != NULL)
     filesizearray[currentline++] = getLlong(linebuf);
   currentline=0;
 }

 if (printfile != NULL) {
   printfileptr=fopen(printfile,"w");
   if (printfileptr==NULL) {fprintf(stderr,"Could not open file %s\n",printfile); exit (EXIT_FAILURE);}
 }


 if (outputdir != NULL)  checkdir(outputdir);

 int cont=1;
 do {
   switch (distrib) {
   case STATIC:        size = staticsize;                              break;
   case UNIFORM:       size = round(unif(startlimit,endlimit));        break;
   case NORMAL:        size = round(norm (meansize,stddev));           break;
   case TRUNC_EXPON:   size =1;                                        break;
   case EXPON:         size = round(expon(meansize));                  break;
   case PARETO:        size = round(pareto(meansize,shape));           break;
   case BOUNDED_PARETO:size = round(bpareto(meansize,shape,endlimit)); break;
   case FILE_SEQ:      size = filesizearray[currentline++];
      if (currentline >= sizelines) currentline =0;                    break;
   case FILE_RAND:
     do {pos=floor(rand_val(0)*sizelines);} while (pos > sizelines-1);  //To handle case where rand_val=1.0
     size=filesizearray[pos];                                          break;
   case END_MARKERD:   fprintf(stderr,"Please specify distribution to generate\n"); exit(1);
     break;
   }

   switch (amounttype) {
   case NRFILES:    if (generatedfiles >= numberoffiles) goto nofile;         break;
   case MIN_SIZE:   if (generatedsize + size >= totalfilesize) cont=0;        break;
   case MAX_SIZE:   if (generatedsize + size > totalfilesize) goto nofile;    break;
   case EXACT_SIZE:
     if (generatedsize + size == totalfilesize) cont=0;
     if (generatedsize + size > totalfilesize) {
fprintf (stderr,"Note: Last generated value truncated! Should be %lld, was truncated to %lld.\n",
 size,totalfilesize - generatedsize);
size = totalfilesize - generatedsize;
cont = 0;
     }
     break;
   case END_MARKERA:  fprintf(stderr,"Please specify amount to generate\n"); exit(1);
break;
   }

   if (size<0)  {fprintf(stderr,"Size > 0 : %10lld, Check parameterization\n",size); exit (EXIT_FAILURE);}

   char *tmpptr=NULL;
   if (outputdir != NULL)   tmpptr = write_file (outputdir,size);
   if (printfile != NULL)   fprintf (printfileptr,"%10lld   %s\n",size,tmpptr == NULL ? " " : tmpptr);
   generatedfiles++;
   generatedsize += size;

   if (debug) {fprintf (stderr,"File#:%8lld  Size:%10lld Generatedsize:%8lld\n",generatedfiles,size,generatedsize);  }
 } while (cont);
nofile:
 return (EXIT_SUCCESS);
}

char *fillchunk_old (char *writebufferidx, unsigned int *chunknrp, int thischunksize)
{
 int k;

 strncpy (writebufferidx,"FIVB",4);
 *(unsigned int*)(writebufferidx+4) = (unsigned int) filenumber;
 *(unsigned int*)(writebufferidx+8) = (unsigned int) (*chunknrp)++;
 *(unsigned int*)(writebufferidx+12) = (unsigned int) thischunksize;
 for (k=16; k<thischunksize-4;k++) *(writebufferidx+k)='J';
 strncpy (writebufferidx+k,"FIVE",4);
 return writebufferidx+k+4;
}

char *fillchunk (char *writebufferidx, unsigned int *chunknrp, int thischunksize)
{
 int k;

 strncpy (writebufferidx,"FIVB               ",16);
 snprintf (writebufferidx+16,16,"Filenr:%8u",filenumber);    //DO NOT CHANGE !!
 snprintf (writebufferidx+32,16,"Chnknr:%8u",(*chunknrp)++); //SCANF USES FORMAT
 snprintf (writebufferidx+48,16,"Chnksz:%8u",thischunksize); //IN findGenFrag!!

 for (k=64; k<thischunksize-4;k++) *(writebufferidx+k)='J';
 strncpy (writebufferidx+k,"FIVE",4);
 return writebufferidx+k+4;
}


char *write_file (char *outputdir, long long filesize) {
 unsigned int i,j,k,retval;
 FILE *fp;
 unsigned int chunknr=0;
 static char filename[2048]={""};       // Output file name string
 static char writebuffer[BUFFERSIZE];                // Buffer
 char *writebufferp=writebuffer;

 if (outputdir[strlen(outputdir)-1] == '/')
   snprintf(filename,512,"%s%s%d",outputdir,"genfile_",filenumber);
 else
   snprintf(filename,512,"%s%s%d",outputdir,"/genfile_",filenumber);

 if (debug) fprintf(stderr," %10lld  %s\n",filesize, filename);

 if ((fp = fopen(filename, "wb")) == NULL){
   fprintf(stderr,"ERROR in creating output file (%s) Aborting.\n", filename);
   exit(1);
 }

 // If text, preload buffer once.
 if (filltype == TEXT){
   k=strlen(fillstring);
   while (writebufferp-writebuffer+k < BUFFERSIZE) {
     strcpy (writebufferp,fillstring);
     writebufferp += k;
   }
   strncpy (writebufferp, fillstring,BUFFERSIZE - (writebufferp - writebuffer));
 }

 //Fill full buffers

 for(i=filesize/BUFFERSIZE; i>0; i--) {
   if (filltype == RANDOM) {  // TODO: Additional fill VARIANT ;RANDOM;RAND BLOCK COPY??
```

```
     for (j=0; j<BUFFERSIZE; j+=2) *(unsigned short*)(writebuffer+j) = (unsigned short)(random()%65535);
   } else if (filltype == CHUNK) {     //Make chunkheaders
     do {
writebufferp=fillchunk (writebufferp,&chunknr,chunksize);
     } while (writebufferp-writebuffer < BUFFERSIZE);    //N.B  BUFFERSIZE % CHUNKSIZE must be 0
   } else if (filltype == TEXT){ //Already preloaded
   } else {
     fprintf(stderr,"No filltype defined....");
     exit(1);
   }
   if (fwrite(writebuffer, 1, BUFFERSIZE, fp ) != BUFFERSIZE) {
     fprintf(stderr,"ERROR when writing to file (%s) \n", filename);
     return filename;
   }
   writebufferp = writebuffer;
 }

 //Fill last partial buffer

 if (filltype == RANDOM) {
   for (j=0; j<filesize%BUFFERSIZE; j+=2) *(unsigned short*)(writebuffer+j) = (unsigned short)(random()%65535);
if (debug) fprintf(stderr," fillrandom %d\n",RAND_MAX);
 } else if (filltype == CHUNK) {     //Make chunkheaders
   while (writebufferp-writebuffer+chunksize < filesize%BUFFERSIZE) {
     writebufferp=fillchunk (writebufferp,&chunknr,chunksize);
   };
   writebufferp=fillchunk (writebufferp,&chunknr,(filesize-(writebufferp-writebuffer))%chunksize);
 } else if (filltype == TEXT){ //Already preloaded
 } else {
     fprintf(stderr,"No filltype defined....");
     exit(1);
   }

 if ((retval=fwrite(writebuffer, 1, filesize%BUFFERSIZE, fp )) != filesize%BUFFERSIZE) {
   fprintf(stderr,"ERROR when writing to file (%s) Wanted %lld, Got %u\n", filename, filesize%BUFFERSIZE, retval);
   return filename;
 }

 fclose(fp);
 filenumber++;
 return filename;
}

int fileoutput_oldgenfile(int argc, char** argv){

return 0;
}



/*
void dummy (void){
 int i,num_values=1;
  double exp_rv  ;
 // Generate and output exponential random variables
 for ( i=0; i<num_values; i++)
 {
   exp_rv = expon(1.0 / lambda);
   fprintf(fp, "%f \n", exp_rv);
 }

}


*/
```

```
//CHECKME!!!!!!!!!!! AND BPARETO OUTPUT
//=========================================================================
//=  Function to generate Pareto distributed RVs using                   =
//=    - Input:  a and k                                                  =
//=    - Output: Returns with Pareto RV                                   =
//=========================================================================
double pareto(double k, double a)
{
 double z;     // Uniform random number from 0 to 1
 double rv;    // RV to be returned

 // Pull a uniform RV (0 < z < 1)
 do
 {
   z = rand_val(0);
 }
 while ((z == 0) || (z == 1));

 // Generate Pareto rv using the inversion method
 rv = k / pow(z, (1.0 / a));

 return(rv);
}


//=========================================================================
//=  Function to generate bounded Pareto distributed RVs using           =
//=    - Input:  a, k, and p                                              =
//=    - Output: Returns with bounded Pareto RV                           =
//=========================================================================
double bpareto(double k, double a, double p)
{
 double z;     // Uniform random number from 0 to 1
 double rv;    // RV to be returned

 // Pull a uniform RV (0 < z < 1)
 do
 {
   z = rand_val(0);
 }
 while ((z == 0) || (z == 1));

 // Generate the bounded Pareto rv using the inversion method
 rv = pow((pow(k, a) / (z*pow((k/p), a) - z + 1)), (1.0/a));

 return(rv);
}


//=========================================================================
//=  Function to generate normally distributed random variable using the =
//=  Box-Muller method                                                    =
//=    - Input: mean and standard deviation                               =
//=    - Output: Returns with normally distributed random variable        =
//=========================================================================
double norm(double mean, double std_dev)
{
 double   u, r, theta;           // Variables for Box-Muller method
 double   x;                     // Normal(0, 1) rv
 double   norm_rv;               // The adjusted normal rv

 // Generate u
 u = 0.0;
 while (u == 0.0)
   u = rand_val(0);
 // Compute r
 r = sqrt(-2.0 * log(u));

 // Generate theta
 theta = 0.0;
 while (theta == 0.0)
   theta = 2.0 * M_PI * rand_val(0);

 // Generate x value
 x = r * cos(theta);

 // Adjust x value for specified mean and variance
 norm_rv = (x * std_dev) + mean;

 // Return the normally distributed RV value
 return(norm_rv);
}




//===========================================================================
//=  Function to generate exponentially distributed random variables        =
//=    - Input:  Mean value of distribution                                 =
//=    - Output: Returns with exponentially distributed random variable     =
//===========================================================================
double expon(double x)
{
 double z;                     // Uniform random number (0 < z < 1)
 double exp_value;             // Computed exponential value to be returned

 // Pull a uniform random number (0 < z < 1)
 do
 {
   z = rand_val(0);
 }
 while ((z == 0) || (z == 1));

 // Compute exponential random variable using inversion method
 exp_value = -x * log(z);

 return(exp_value);
}

//===========================================================================
//=  Function to generate uniformly distributed random variables            =
//=    - Input:  Min and max values                                         =
//=    - Output: Returns with uniformly distributed random variable         =
//===========================================================================
double unif(double min, double max)
{
 double z;                     // Uniform random number (0 < z < 1)
 double unif_value;            // Computed uniform value to be returned

 // Pull a uniform random value (0 < z < 1)
 z = rand_val(0);

 // Compute uniform continuous random variable using inversion method
 unif_value = z * (max - min) + min;

 return(unif_value);
}
```

```
/* CHECKME: Seems to give non-random numbers for too large seeds */
/*
[johan@localhost FragEvalTools]$ ./a.out -d 0 -n 100 20   -c 10000 -w test.txt -r 422343534454345519; awk '{tot=tot+=43} END
1000000
*/
/* Also, should resolution be 64 bit-based instead of 32 bit? */
//=========================================================================
//= Multiplicative LCG for generating uniform(0.0, 1.0) random numbers    =
//=   - From R. Jain, "The Art of Computer Systems Performance Analysis," =
//=     John Wiley & Sons, 1991. (Page 443, Figure 26.2)                  =
//=========================================================================
double rand_val(int seed)
{
 const long  a =      16807;  // Multiplier
 const long  m = 2147483647;  // Modulus
 const long  q =     127773;  // m div a
 const long  r =       2836;  // m mod a
 static long x;               // Random int value
 long        x_div_q;         // x divided by q
 long        x_mod_q;         // x modulo q
 long        x_new;           // New x value

 // Set the seed if argument is non-zero and then return zero
 if (seed > 0)
 {
   x = seed;
   return(0.0);
 }

 // RNG using integer arithmetic
 x_div_q = x / q;
 x_mod_q = x % q;
 x_new = (a * x_mod_q) - (r * x_div_q);
 if (x_new > 0)
   x = x_new;
 else
   x = x_new + m;

 // Return a random value between 0.0 and 1.0
 return((double) x / m);
}



void checkdir (char *outputdir)
{
 struct stat sb;

 if (stat(outputdir, &sb) == -1) {
   perror("Parsing output dir");
   exit(EXIT_FAILURE);
 }

 if ((sb.st_mode & S_IFMT) != S_IFDIR) {
   printf("Unexepected file type for output directory:    ");
   switch (sb.st_mode & S_IFMT) {
   case S_IFBLK:  printf("block device\n");            break;
   case S_IFCHR:  printf("character device\n");        break;
   case S_IFIFO:  printf("FIFO/pipe\n");               break;
   case S_IFLNK:  printf("symlink\n");                 break;
   case S_IFREG:  printf("regular file\n");            break;
   case S_IFSOCK: printf("socket\n");                  break;
   default:       printf("unknown?\n");                break;
   }
   exit(EXIT_FAILURE);
 }
}
```

A.3 Python Scripts

A.3.1 script2.py

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

# written for python 2.5 & cygwin compatibility
# By Zak Blacher - 2010

from subprocess import Popen, PIPE, list2cmdline, call
from os import fork
from os.path import abspath

from time import time, sleep
import sqlite3

# --- EXECUTION PARAMETERS ---

start           = 0
# Where to start in the iteration process (for resuming)

path_to_exec    = "./"
# Path to C executables

max_iter_time   = 60*60 # 1 hour
ps_check_int    = 5     # check every 5 seconds
# parameters for process babysitting

max_genfile_amt = "1Gb"
# generate this much data
output_file     = "collection.db"
# database for storing collected information
table_name      = "cluster_size_distr_all"

# table in database for this test


dd_buffer_size  = '16MB'
# dd buffer size

fs_params = {
  'name'     : 'haystack',
  'dev'      : '/dev/sda2',
  'path'     : '/cygdrive/d',
  'win_path' : 'D:'
}
# target filesystem parameters

# --- TEST PARAMETERS ---
reps = 5
fs_types       = ['ntfs']
cluster_sizes  = ['1024','2048','4096','8192', '16k','32k']

# Offending File Generation Parameters
of_params = [
 ('1000 files 250 KbKi      , ['-s', '250KbKi-c', '1000']),
]

# Random File Generation Parameters
rf_params = [
 ('Exponential: 141 KbKi    , ['-e', '141KbKi0.00006']),
 ('Exponential: 40 KbKi     , ['-e', '40KbKi0.00006']),
 ('Exponential: 80 KbKi     , ['-e', '80KbKi0.00006']),
 ('Uniform: 10-30 KbKi      , ['-u', '10KbKi30KbKi
 ('Uniform: 2.4-3.6 Mbyte'    , ['-u', '2458KbKi3686KbKi
 ('Uniform: 20-60 KbKi      , ['-u', '20KbKi60KbKi
 ('Uniform: 4-12 KbKi       , ['-u', '4KbKi12KbKi
 ('Uniform: 40-120 KbKi     , ['-u', '40KbKi120KbKi
 ('Uniform: 400-1200 KbKi   , ['-u', '4000KbKi1200KbKi
]

####################################################################
#Here Be Dragons

def list3cmdline(s):
  # print and flatten command line
  s = list2cmdline(s)
  print s
  return s

def format_permute(start=0):
  # generator for parameter permutations
  count = 0
  for i in range(reps):
    for j in fs_types:
      for k in cluster_sizes:
        for (l,m) in of_params:
          for (n,p) in rf_params:
            if (count >= start):
              yield (i,j,k,(l,m),(n,p))
            count += 1

print "Performing",len(cluster_sizes)*len(fs_types)*len(of_params)*len(rf_params)*reps,"Tests..."


# try to create table
dbconn = sqlite3.connect(output_file)
cursor = dbconn.cursor()

try:
  cursor.execute('create table '+table_name+' (rep,fs_type,cluster_size, \
  of_param,rf_param,b_nrfrag,b_procbyte,b_slackbKi \
  b_compchunk,b_icompchunk,a_nrfrag,a_procbyte,a_slackbKia_compchunk,a_icompchunk)')
  # repetition, filesystem type, cluster size, original file parameters, random file parameters,
  # before {number of fragments, processed bytes, slack bytes, complete chunks, incomplete chunks}
  # after {number of fragments, processed bytes, slack bytes, complete chunks, incomplete chunks}
except:
  pass


for (rep,fs_type,cluster_size,(of_key,of_param),(rf_key,rf_param)) in format_permute(start):
  print [rep,fs_type,cluster_size,(of_key,of_param),(rf_key,rf_param)]

  try:
    start_time = time()
    # 'Zeroing Disk.'
    call(list3cmdline(['dd', 'if=/dev/zero','of='+fs_params['dev'],'bs='+dd_buffer_size]),shell=True)

    # 'Formatting drive'
    call(("format.com %s /FS:%s /V:%s /A:%s /X /Q /Y" % (fs_params['win_path'],fs_type,
      fs_params['name'],cluster_size)),shell=True)
    # 'Counting Initial Fragments'
    ps = Popen([path_to_exec+'findGenFrag.exe','-m','-c',fs_params['dev']],stdout=PIPE)

    while(ps.poll() == None and time() <= (start_time + max_iter_time)):
      sleep(ps_check_int)

    if (time() > (start_time + max_iter_time)):
      ps.kill()
      raise Exception('timeout','initial')

    s = ps.communicate()[0]
    (b_nrfrag,b_procbyte,b_slackbKib_compchunk,b_icompchunk) = s.splitlines()[0].split(';')

    # b_nrfrag, 'fragments initially found. Generating Files'
    ps = Popen([path_to_exec+'genDistrFile.exe','-o',fs_params['path']]+of_param)

    while(ps.poll() == None and time() <= (start_time + max_iter_time)):
      sleep(ps_check_int)

    if (time() > (start_time + max_iter_time)):
      ps.kill()
      raise Exception('timeout','initial generation')

    # 'Deleting Generated Files'
    call(list3cmdline(['rm','-r','-s',fs_params['path']+'/genfile_*']),shell=True)

    #print 'Writing Random Content Files'

    ps = Popen([path_to_exec+'genDistrFile.exe','-q','-m',max_genfile_amt,'-o',fs_params['path']]+rf_param)

    while(ps.poll() == None and time() <= (start_time + max_iter_time)):
      sleep(ps_check_int)

    if (time() > (start_time + max_iter_time)):
      ps.kill()
      raise Exception('timeout','random generation')

    # 'Counting Fragments'
    ps = Popen([path_to_exec+'findGenFrag.exe','-m','-c',fs_params['dev']],stdout=PIPE)

    while(ps.poll() == None and time() <= (start_time + max_iter_time)):
      sleep(ps_check_int)

    if (time() > (start_time + max_iter_time)):
      ps.kill()
      raise Exception('timeout','final count')

    s = ps.communicate()[0]
    (a_nrfrag,a_procbyte,a_slackbKia_compchunk,a_icompchunk) = s.splitlines()[0].split(';')

    # a_nrfrag, 'fragments found.'
    cursor.execute('insert into '+table_name+' values (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)',[rep,fs_type,cluster_size,
      of_key,rf_key,b_nrfrag,b_procbyte,b_slackbKi
      b_compchunk,b_icompchunk,a_nrfrag,a_procbyte,a_slackbKia_compchunk,a_icompchunk])
    # 'added record tuple to database'

    dbconn.commit()

  except:
    print "WARNING: COULD NOT EXECUTE COMMANDS FOR TUPLE",[rep,fs_type,cluster_size,of_key,rf_key,start,end]
```

A.3.2 script_rh.py

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

# written for python 2.5 & cygwin compatibility
# By Zak Blacher - 2010

from subprocess import Popen, PIPE, list2cmdline, call
from os import fork
from os.path import abspath

from time import time, sleep
import sqlite3

# --- EXECUTION PARAMETERS ---

start           = 0
# Where to start in the iteration process (for resuming)

path_to_exec    = "./"
# Path to C executables
gen_file_dir = 'genfiles/'
# Path for stored random files

max_iter_time   = 60*60 # 1 hour
ps_check_int    = 5     # check every 5 seconds
# parameters for process babysitting

max_genfile_amt = "1Gb"
# generate this much data
output_file     = "collection.db"
# database for storing collected information
table_name      = "size_distr_test_hash"
# table in database for this test


dd_buffer_size  = '16MB'
# dd buffer size

fs_params = {
  'name'     : 'haystack',
  'dev'      : '/dev/sda2',
  'path'     : '/cygdrive/d',
  'win_path' : 'D:'
}
# target filesystem parameters

# --- TEST PARAMETERS ---
reps = 5
fs_types = ['ntfs']
cluster_sizes = ['4096']
of_params = [
 ('1000 files 250 KbKi  , ['-s', '250KbKi-c', '1000']),
]
rf_params = [
 ('Uniform: 2.4-3.6 Mbyte b'    , ['-u', '2457KbKi3686KbKi
]

####################################################################
#Here Be Dragons

def list3cmdline(s):
  # print and flatten command line
  s = list2cmdline(s)
  print s
  return s

def format_permute(start=0):
  # generator for parameter permutations
  count = 0
  for i in range(reps):
    for j in fs_types:
      for k in cluster_sizes:
        for (l,m) in of_params:
          for (n,p) in rf_params:
            if (count >= start):
              yield (i,j,k,(l,m),(n,p))
            count += 1

print "Performing",len(cluster_sizes)*len(fs_types)*len(of_params)*len(rf_params)*reps,"Tests..."


# try to create table
dbconn = sqlite3.connect(output_file)
cursor = dbconn.cursor()

try:
  cursor.execute('create table '+table_name+' (rep,fs_type,cluster_size,of_param,rf_param,hashmatch)')
except:
  pass

# generate bloom filter

if (start == 0):
  # new data set. generate known content files
  call(('mkdir -p ./'+gen_file_dir),shell=True)
  call(('rm '+gen_file_dir+'/*'),shell=True)
  call(('rm bloomfilter.bin metadata.bin'),shell=True)

  ps = Popen(list3cmdline([path_to_exec+'genDistrFile.exe','-q','-o',gen_file_dir]+of_params[0][1]),shell=True)

  # monitor creation process
  start_time = time()
  while(ps.poll() == None and time() <= (start_time + max_iter_time)):
    sleep(ps_check_int)

  if (time() > (start_time + max_iter_time)):
    ps.kill()
    raise Exception('timeout','initial generation')

# generate bloom filter file
call((path_to_exec+'frag_find_v7.exe --create -v 0 --rolling-hash "'+abspath(gen_file_dir)+'"'),shell=True)

for (rep,fs_type,cluster_size,(of_key,of_param),(rf_key,rf_param)) in format_permute(start):
  # Each individual test run executes in this loop.

  # print [rep,fs_type,cluster_size,(of_key,of_param),(rf_key,rf_param)]

  try:
    # each test run is aborted if it takes longer than max_iter_time seconds
    start_time = time()
    # print 'Zeroing Disk.'
    call(list3cmdline(['dd', 'if=/dev/zero','of='+fs_params['dev'],'bs='+dd_buffer_size]),shell=True)

    # print 'Formatting drive'
    call(("format.com %s /FS:%s /V:%s /A:%s /X /Q /Y" % (fs_params['win_path'],fs_type,fs_params['name'],
     cluster_size)),shell=True)

    # print 'Copying known files'
    call('cp '+gen_file_dir+'/* '+fs_params['path'],shell=True)
    # print 'Deleting Generated Files'
    call(list3cmdline(['rm','-r','-s',fs_params['path']+'/genfile_*']),shell=True)

    # print 'Writing over with Structured Content Files'
    ps = Popen([path_to_exec+'genDistrFile.exe','-m',max_genfile_amt,'-o',fs_params['path']]+rf_param)
    while(ps.poll() == None and time() <= (start_time + max_iter_time)):
      sleep(ps_check_int)

    if (time() > (start_time + max_iter_time)):
      ps.kill()
      raise Exception('timeout','random generation')

    # CYGWIN PROBLEM: cannot call frag_find on a device image. must first copy to .iso file.
    call(list3cmdline(['dd','if='+fs_params['dev'], 'of='+fs_params['name'],'bs='+dd_buffer_size]),shell=True)

    # print 'Counting Hash Matches'
    ps = Popen([path_to_exec+'frag_find_v7.exe', '--scan', '-v', '0', '--rolling-hash', '--sort-by-file',
               '--output-file-coverage',fs_params['name']],stdout=PIPE)
    while(ps.poll() == None and time() <= (start_time + max_iter_time)):
      sleep(ps_check_int)

    if (time() > (start_time + max_iter_time)):
      ps.kill()
      raise Exception('timeout','final count')

    s = ps.communicate()[0]
    hashmatch = s.splitlines()[0].strip()

    # add test run data to database
    cursor.execute('insert into '+table_name+' values (?,?,?,?,?,?)',[rep,fs_type,cluster_size,of_key,rf_key,hashmatch])
    dbconn.commit()

  except:
    print "WARNING: COULD NOT EXECUTE COMMANDS FOR TUPLE",[rep,fs_type,cluster_size,of_key,rf_key]
```

A.4 Extension & Misc Functions

A.4.1 OpenOffice Graph Export Macro

```
   ' Export all charts from a Calc spreadsheet -- Jose Fonseca
   ' Updated as standalone by Zak Blacher 2010

Sub ExportGraphs
   Dim oDoc, oDocCtrl, oDocFrame, oDispatchHelper
   oDoc = ThisComponent
   oDocCtrl = oDoc.getCurrentController()
   oDocFrame = oDocCtrl.getFrame()
   oDispatchHelper = createUnoService( "com.sun.star.frame.DispatchHelper" )

   Dim storeUrl
   storeUrl = oDoc.getURL()
   storeUrl = Left( storeUrl, Len( storeUrl ) - 4 )

   nCharts = 0

   ' Search the draw page for the chart.
   Dim oSheets, oSheet, oDrawPage, oShape
   oSheets = oDoc.getSheets()
   For i = 0 to oSheets.getCount() - 1
      oSheet = oSheets.getByIndex( i )
      oDrawPage = oSheet.getDrawPage()
      For j = 0 to oDrawPage.getCount() - 1
         oShape = oDrawPage.getByIndex( j )
         ' Can't call supportsService unless the com.sun.star.lang.XServiceInfo is present.
         If HasUnoInterfaces( oShape, "com.sun.star.lang.XServiceInfo" ) Then
            If oShape.supportsService( "com.sun.star.drawing.OLE2Shape" ) Then
               ' Is it a Chart?
               If oShape.CLSID = "12DCAE26-281F-416F-a234-c3086127382e" Then
                  ' Select the chart shape.
                  oDocCtrl.select( oShape )
                  oDispatchHelper.executeDispatch( oDocFrame, ".uno:Copy", "", 0, Array() )
                  ' export the chart
                  nCharts = nCharts + 1
                  ExportSelection( storeUrl + "_chart" + nCharts + ".eps", "image/x-eps" )
               EndIf
            EndIf
         EndIf
      Next
   Next
End Sub

Sub ExportSelection(url As String, mediaType As String)
   ' Create a new Draw document
   Dim aArgs(1) As New com.sun.star.beans.PropertyValue
   aArgs(0).Name = "Hidden"
   aArgs(0).Value = True
   oDrawDoc = StarDesktop.loadComponentFromURL( "private:factory/sdraw", "_blank", 0, aArgs() )

   ' Past current selection
   Dim oDrawDocCtrl, oDrawDocFrame, oDispatchHelper
   oDrawDocCtrl = oDrawDoc.getCurrentController()
   oDrawDocFrame = oDrawDocCtrl.getFrame()
   oDispatchHelper = createUnoService( "com.sun.star.frame.DispatchHelper" )
   oDispatchHelper.executeDispatch( oDrawDocFrame, ".uno:Paste", "", 0, Array() )

   ' Get an export filter object
   Dim exportFilter
   exportFilter = createUnoService( "com.sun.star.drawing.GraphicExportFilter" )

   ' get first draw page
   Dim oDrawPages, oDrawPage, oShape
   oDrawPages = oDrawDoc.getDrawPages()
   oDrawPage = oDrawPages.getByIndex( 0 )
   oShape = oDrawPage.getByIndex( 0 )
   exportFilter.setSourceDocument( oShape )

   ' Set the filter data
   Dim aFilterData(5) As New com.sun.star.beans.PropertyValue
   aFilterData(0).Name = "Level" '1=PS level 1, 2=PS level 2
   aFilterData(0).Value = 2
   aFilterData(1).Name = "ColorFormat" '1=color, 2=grayscale
   aFilterData(1).Value = 1
   aFilterData(2).Name = "TextMode" '0=glyph outlines, 1=no glyph outlines, see ooo bug 7918
   aFilterData(2).Value = 1
   aFilterData(3).Name = "Preview" '0=none, 1=TIFF, 2=EPSI, 3=TIFF+EPSI
   aFilterData(3).Value = 0
   aFilterData(4).Name = "CompressionMode" '1=LZW, 2=none
   aFilterData(4).Value = 2

   Dim aProps(2) As New com.sun.star.beans.PropertyValue
   aProps(0).Name = "MediaType"
   aProps(0).Value = mediaType
   aProps(1).Name = "URL"
   aProps(1).Value = url
   aProps(2).Name = "FilterData"
   aProps(2).Value = aFilterData()

   exportFilter.filter( aProps() )
End Sub
```

A.4.2 extension-functions.c

(This file was obtained from 'http://www.sqlite.org/contrib/download/extension-functions.c?get=25', and has not been printed.)