NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

USING DISTINCT SECTORS IN MEDIA SAMPLING AND FULL MEDIA ANALYSIS TO DETECT PRESENCE OF DOCUMENTS FROM A CORPUS

by

Kristina Foster

September 2012

Thesis Advisor:	Simson Garfinkel
Second Reader:	Neal Ziring

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 11–9–2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* 2011-03-28—2012-09-21

4. TITLE AND SUBTITLE

Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Kristina Foster

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

Forensics examiners frequently search for known content by comparing each file from a target media to a known file hash database. We propose using sector hashing to rapidly identify content of interest. Using this method, we hash 512 B or 4 KiB disk sectors of the target media and compare those to a hash database of known file blocks, fixed-sized file fragments of the same size. Sector-level analysis is fast because it can be parallelized and we can sample a sufficient number of sectors to determine with high probability if a known file exists on the target. Sector hashing is also file system agnostic and allows us to identify evidence that a file once existed even if it is not fully recoverable. In this thesis we analyze the occurrence of distinct file blocks–blocks that only occur as a copy of the original file–in three multi-million file corpora and show that most files, including documents, legitimate and malicious software, consist of distinct blocks. We also determine the relative performance of several conventional SQL and NoSQL databases with a set of one billion file block hashes.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	87	

THIS PAGE INTENTIONALLY LEFT BLANK

USING DISTINCT SECTORS IN MEDIA SAMPLING AND FULL MEDIA ANALYSIS TO DETECT PRESENCE OF DOCUMENTS FROM A CORPUS

Kristina Foster
Civilian
B.S., Computer Science and Electrical Engineering, MIT, 2003
M.Eng., Electrical Engineering and Computer Science, MIT, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2012

Author:	Kristina Foster
Approved by:	Simson Garfinkel Thesis Advisor
	Neal Ziring Second Reader
	Peter Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Forensics examiners frequently search for known content by comparing each file from a target media to a known file hash database. We propose using sector hashing to rapidly identify content of interest. Using this method, we hash 512 B or 4 KiB disk sectors of the target media and compare those to a hash database of known file blocks, fixed-sized file fragments of the same size. Sector-level analysis is fast because it can be parallelized and we can sample a sufficient number of sectors to determine with high probability if a known file exists on the target. Sector hashing is also file system agnostic and allows us to identify evidence that a file once existed even if it is not fully recoverable. In this thesis we analyze the occurrence of distinct file blocks–blocks that only occur as a copy of the original file–in three multi-million file corpora and show that most files, including documents, legitimate and malicious software, consist of distinct blocks. We also determine the relative performance of several conventional SQL and NoSQL databases with a set of one billion file block hashes.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

List of Acronyms and Abbreviations

ASCII	American Standard Code for Information Interchange
b	bit
B	Byte
BF	Bloom Filter
BtrFS	B-tree File System
CRC32	Cyclic Redundancy Check (32 bits)
CRC64	Cyclic Redundancy Check (64 bits)
DBMS	Database Management System
EXT3	Third Extended File System
EXT4	Fourth Extended File System
FBHMA	File Based Hash Map Analysis
FAT	File Allocation Table
GB	Gigabyte (10^9 bytes)
GiB	Gibibyte (2^{30} bytes)
Govdocs1	Million Government Document corpus
html	HTML file extension
HTML	HyperTExt Markup Language file format
jpg	JPEG file extension
JPEG	Joint Photographic Experts Group file format
K	Thousand (10^3)
KB	Kilobyte (10^3 bytes)
KiB	Kibibyte (2^{10} bytes)
M	Million (10^6)
MB	Megabyte (10^6 bytes)

MD5	MD5 message digest algorithm
MiB	Mebibyte (2^{20} bytes)
OCMalware	Offensive Computing Malware corpus
OS	Operating System
NOOP	No Operation assembly instruction
NoSQL	“Not only SQL” model for non-relational database management
NSRL	National Software Reference Library
NSRL RDS	NSRL Reference Data Set
NTFS	New Technology File System
pdf	PDF file extension
PDF	Adobe Portable Document Format
PE	Portable Executable file format
RAM	Random access memory
RPC	Remote procedure call
SHA-1	Secure Hash Algorithm, version 1
SHA-3	Secure Hash Algorithm, version 3
SQL	Structured Query Language for relational database management
TB	Terabyte (10^{12} bytes)
TiB	Tebibyte (2^{40} bytes)
TPS	Transactions per Second
txt	TXT file extension
TXT	ASCII text file
UPX	The Ultimate Packer for eXecutables

Acknowledgements

I would like to thank Dr. Simson Garfinkel for his outstanding mentorship. I am grateful for his guidance throughout the thesis process, the opportunities he made available to share my work with the community and his strong example of leadership and technical expertise.

I would also like to thank Mr. Neal Ziring for being my second reader and providing careful insight and feedback on the thesis. Mr. Ziring has been a long time unofficial mentor of mine and I appreciate his willingness to invest his time and impart invaluable guidance and advice.

I would like to thank Dr. Joel Young for his guidance and willingness to assist me in various parts of my thesis. His instruction in experiment design and database performance was critical to obtaining sound and accurate results.

Thank you to all of the professors who provided instruction and guidance during my time at NPS. My experience was greatly enriched through our interactions.

Finally, I would like to thank my mother, Berdia, and fiance, Dewey. I thank God for such a strong support system that keeps me focused and encouraged. I could not do it without you all.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

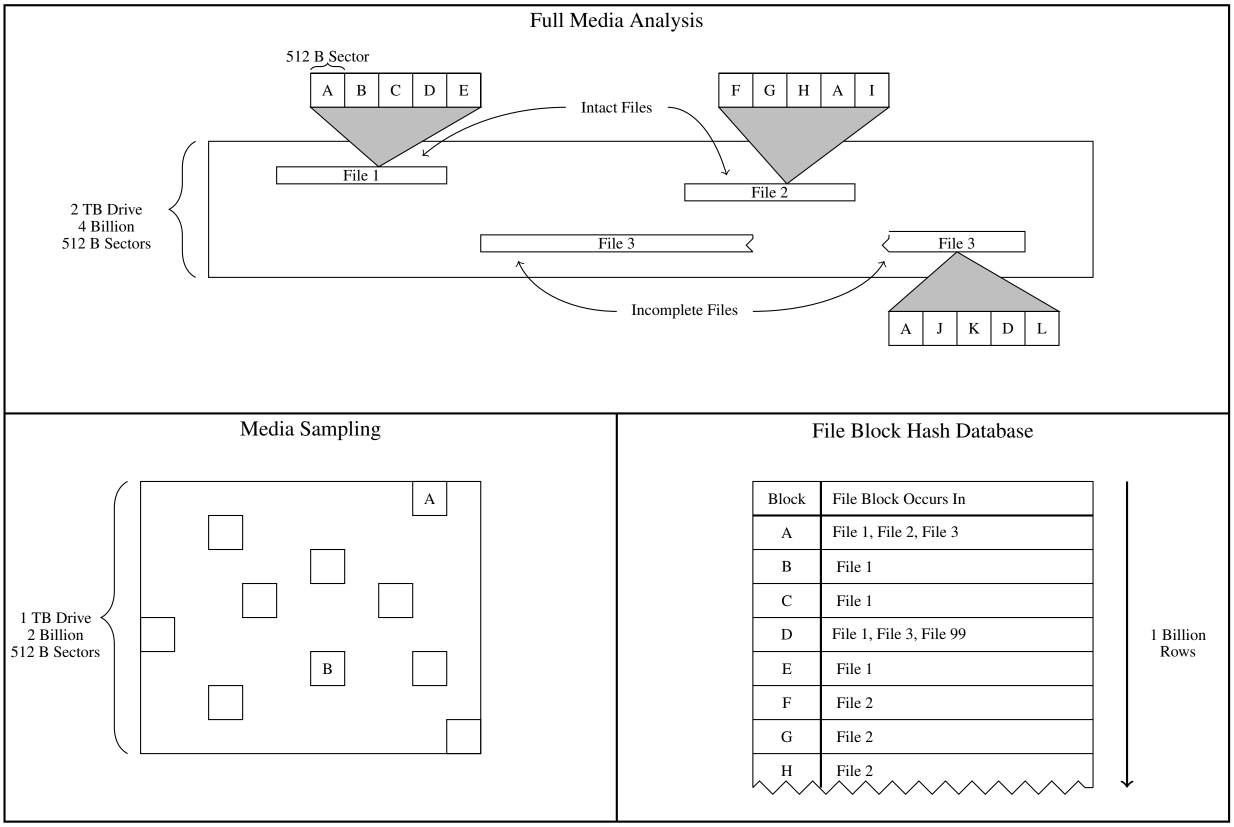
Quickly identifying content of interest on digital media is critical to the forensic investigation process. Given a large disk or set of disks, an examiner requires an efficient triage process to determine if known content is present. Today, examiners identify content by comparing files stored on the target media to a database of known file hashes collected from previous investigations. Traditional forensic tools identify stored files by analyzing the file system or carving files based on headers and footers [1–4].

Analyzing the file system to identify content has several shortcomings. Relevant content may be stored in areas that are not directly parsed by the file system, such as unallocated or slack space. Portions of files or the file system may be unreadable due to partial overwriting or media failure. The current methods are not robust to new or unknown data types, file formats or file systems. Finally, the entire file system must be read to find files of interest and the search process is difficult to parallelize due to the file system tree structure [5].

Although file carving addresses some of these issues by parsing the raw bytes on the disk, carving itself has several shortcomings. For example, file carving based on headers and footers is not effective at identifying content from overwritten or partially destroyed files, or content that is fragmented into multiple locations on the media. File carving is also prone to false positives.

1.1 How distinct sector identification works

We propose a forensic method that uses sector hashing to quickly identify content of interest. Using this method we search for content in disk sectors, fixed-sized chunks of physical disk that are the smallest unit to which data can be written. Current file systems such as FAT, NTFS, Ext3 and Ext4 and next generation file systems such as ZFS and the B-tree File System (BtrFS) write files on sector boundaries. The standard sector size is 512 B, although most modern disks are moving to 4 KiB for format efficiency and more robust error correction [6]. For example, when a 60 KB JPEG file is stored, the first 512 B are written to one sector, the second 512 B are written to the next sector and so on. Because most files are sector aligned, we can search for content by comparing the hash of each 512 B or 4 KiB disk sector on the target media to a hash database of fixed-sized file fragments of the same size, which we call file *blocks*.



Block	File Block Occurs In
A	File 1, File 2, File 3
B	File 1
C	File 1
D	File 1, File 3, File 99
E	File 1
F	File 2
G	File 2
H	File 2

Figure 1.1: Because files are stored on sector boundaries, we can search disk sectors for file blocks, or fixed-sized chunks of data equal in size to the disk sectors. We create a file block hash database that contains block hashes for every file that we have ever seen during an investigation. A database with 1 billion 512 B block hashes can reference 476 GB of content. Sector hashing depends on the existence of distinct file blocks, or blocks that only occur as a copy in the original file. With full media analysis, all 4 billion sectors from the 2 TB drive are compared to the file block hash database. With media sampling, only 1 million of the 2 billion sectors from the 1 TB drive are compared to identify a 4 MB file that has all distinct blocks with 98.17% accuracy. If block B is seen on a disk sector, then there is a good chance that File 1 also exists on the disk. Block B only occurs in one file in our large corpus of known files and is effectively distinct. If Block A is seen on a disk sector, then we are not sure if any of the files exist. Block A is non-distinct. Sector hashing can quickly identify fully intact and incomplete files that contain distinct blocks.

This example demonstrates the use of sector hashing to identify the presence of three files (1, 2 & 3) on the subject media. The block hash database contains all of the blocks from a corpus of every file that has ever been seen during an investigation. The database is a key-value store where the key is a hash of a file block and the value is a list of every file in which the block occurs.

Figure 1.1 is a graphical representation of a 2 TB disk that has four billion 512-byte sectors. It contains three previously seen files; File 1, File 2 and File 3. File 1 and File 2 are both 60 KB JPEG images that have 120 512-byte blocks, matching the sector size. The files are intact, which means that every file block is currently stored in a disk sector. As shown in Figure 1.1, File 1 contains blocks A - E and File 2 contains blocks A and F - I.

File 3 is a 4 MB high-resolution JPEG image that has 8,192 blocks. This file is incomplete because some of the sectors that previously stored its blocks have been overwritten. The remaining blocks have not been overwritten and are stored in the disk sectors. This scenario occurs after a file is deleted and the containing sectors are made available for new files. File 3 contains blocks A, D and J - L.

1.1.1 Sector hashing and full media analysis for residual data identification

Using sector hashing in full media analysis, we compare every sector of the 2 TB disk to the block hash database. When the sector that contains block B is identified, we know that File 1 *may* be present on the disk because block B only occurs in File 1 of all the files in our corpus. We do not know if File 1 is *definitely* present because block B could occur in a different file that is not included in our corpus–knowing that block B is present is not sufficient to prove that File 1 is also present. When the sector that contains block A is identified, we have even less confidence that File 1, File 2 or File 3 are present. Since Block A occurs in multiple files in our corpus, we believe the block likely occurs in other files that are not in our corpus–knowing that block A is present does not prove that any of the files from our corpus are present.

Sector hashing for file identification depends on the existence of *distinct* blocks, or blocks that only occur on media as a copy of the original file for all files. We cannot prove that a block is *universally* distinct. However, we can treat blocks that only occur once in a large file corpus, such as block B in our example, as if it were universally distinct. Doing so allows us to quickly find evidence that a file is present on disk. Deeper file-level analysis is used to confirm the file's presence.

After analyzing all of the sectors from our target disk we learn that all of the blocks from File 1 and File 2 and some of the blocks from File 3 are stored on the disk. If the majority of File 1's blocks do not occur in any other corpus file, then the results provide strong evidence that File 1 is currently present on the disk. The same is true for File 2. If File 3's block are also not repeated elsewhere, then the results provide strong evidence that File 3 was once present on the disk.

1.1.2 Sector hashing and random sampling for triage

Using sector hashing for media sampling also allows for a faster triage process. Instead of searching all of the disk sectors, we can search a sample set of randomly chosen sectors to determine with high probability that a file is present as illustrated in Figure 1.1. The sample size must be large enough to ensure that we will almost certainly select a sector that contains at least one of the blocks in the file if it is present. We can determine an appropriate sample size using the well known "urn" problem, a statistical model that describes the probability of pulling some number of red beans out of an urn that contains a mix of red and black beans randomly distributed [7].

The red beans are the sectors that contain the distinct blocks of the content we are trying to identify. The black beans are sectors that do not contain the distinct blocks, all remaining sectors. The total number of beans is the number of sectors on the target media. If we are trying to identify a 4 MB JPEG of all distinct blocks on a 1 TB drive, there are 8,000 red beans (*C*), and 2 billion beans in total (*N*). If we randomly select 1 million (*n*) beans we have a 98.17% chance of selecting a red bean at least once, or detecting the 4 MB file.

Equation 1.1 calculates the probability of not finding even a single red bean in *n* draws and subtracts that from one to get *p*, the probability that at least one red bean is found in *n* draws:

$$p = 1 - \prod_{i=1}^{n} \frac{((N-(i-1))-C)}{(N-(i-1))} \tag{1.1}$$

Using sector hashing with random sampling provides a quick triage method to determine if a file of interest is likely present on a target media. The method is file system and file type agnostic; as long as we can read the disk sectors and have a copy of the file in our block hash database, we can use sector hashing to find evidence of the file on disk. Sector hashing can also find evidence of a file that was once present but has been partially overwritten and is not fully recoverable.

1.2 Design Issues

There are several design issues in implementing media analysis with sector hashing. The first is choosing an appropriate block size for the hash database. We considered both 512 B and 4 KiB, the standard sector sizes for current drives. Using 512 B blocks allows for more granularity in the search but requires that we store eight times as many hashes—there are eight 512 B blocks in every 4 KiB block. As discussed in Chapter 7, the size of the block hash database has a major effect on performance, so it is critical that we maintain the minimum amount of data without missing evidence of a file's presence. The results from the distinct block experiment in Chapter 5 suggest that we do not loose meaningful precision by using a 4 KiB block since there are similar trends in the 512 B and 4 KiB blocks found in millions of real files and various file types.

If we search for 4 KiB blocks on a target media that has 512 B sectors, we can read eight sectors at once to compare to the hash database. We would also have to read multiple 4 KiB chunks from the media that start at different 512 B offsets (e.g. offsets of 0 B, 512 B, 1024 B, etc.) to ensure that we do not miss any 4 KiB blocks due to alignment [5].

Based on storage requirements and because we do not loose meaningful precision, 4 KiB is the appropriate block size.

We chose to use the MD5 hash algorithm to compute block hashes because it is widely used within the forensic community and is computationally fast. We are not concerned that MD5 is no longer collision resistant because our technique relies on using hashes to match known files to target media. If an adversary creates a collision for a set of block hashes then we will still find the file because it will still match hashes in our database. In the future, it would be appropriate to move to SHA-3, as it will probably be faster than MD5.

Next, one must determine an appropriate data storage and query method for the block hash database. The National Software Reference Library (NSRL) Reference Data Set (RDS) is a corpus of standard system files used for forensic investigations. The 2009 RDS contains over twelve million files with an average file size of 240 KB. A database of the 128-bit MD5 hash of every 512 B block for the 2009 RDS requires approximately 92 GB of storage. We would like to query the database as quickly as possible to allow for rapid triage analysis. It is important that our data storage method can handle large volumes of data and can be efficiently queried when analyzing media.

The 92 GB database can be stored in Random Access Memory (RAM), in flash on a Solid State Drive (SSD), or on a spinning magnetic drive. Clearly it is faster to store such a database in RAM. Nevertheless, it makes sense to compare different strategies for organizing the 92 GB database, as even a RAM-based database will have very different performance parameters with different organizations.

We test the relative performance of several conventional SQL and NoSQL Database Management Systems (DBMS) in managing a database of one billion hashes. With a 4 KiB block size, one billion hashes allows us to index 4 TB of content. Our results show that a custom storage solution is required to support the hash lookup speeds that our application requires.

A third issue is determining the occurrences of distinct blocks in our file corpora. We analyze three multi-million file corpora that contain real documents, system files, and legitimate and malicious software. To our knowledge, there are no previous studies analyzing the co-occurrence of blocks across such a large number of files and file types. By using these corpora we can begin to make general conclusions about the true frequency of distinct blocks. Our findings suggest that most files are made up of distinct blocks that identify a single specific file.

We are also interested in determining rules to quickly eliminate disk sectors that store file blocks that are common among many documents or file types. These sectors can be ignored early in the analysis process. Omitting likely non-distinct sectors will improve performance by minimizing the number of disk sectors that are compared to the file block hash database.

The fourth issue is determining the appropriate tool architecture for use in the field. There are numerous restrictions that we consider for deployed operations including limited storage space and computational power.

1.3 Distinct Blocks

Identifying files with sector hashes relies on the presence of distinct file blocks. A *distinct* block is one that does not exist more than once in the universe except as a block in a copy of the original file. Using distinct blocks as a forensic tool leverages two hypotheses [5]:

1. If a block of data from a file is distinct, then a copy of that block found on a data storage device is evidence that the file is or was once present.
2. If the blocks of that file are shown to be distinct with respect to a large and representative corpus, then those blocks can be treated as if they are universally distinct.

The first hypothesis is true by the definition of distinct blocks. If the block only exists as a block in a specific file, then if the block is found on a piece of target media then the file must exist or have previously existed.

The second hypothesis deals with the method of determining if a file block is distinct. It is impossible to prove that a block is universally distinct because doing so would require comparing the block to every block. However, we *can* identify blocks that only appear once in millions of files and treat them as if they were universally distinct in the context of finding possible evidence that a file once existed on a piece of media. Making such a finding requires that we tabulate all the blocks in a sufficiently large corpus that contain the same file types as from which the potentially distinct block came.

We performed this exercise with three large file corpora that each contain millions of files of various file types including documents, operating system files and legitimate and malicious software. We find that the overwhelming majority of the file blocks were distinct with respect to each corpus (and between corpora as well) and could therefore be used to identify a single specific file.

Ideally all files would consist of mostly distinct blocks, or blocks that only occur in one specific file. Finding one distinct block from a file on a target disk is not as convincing as finding multiple distinct blocks from the same file on disk. Furthermore, if we find many distinct blocks from a specific file and if the blocks are stored contiguously, we have higher confidence that the file exists or previously existed.

1.4 Usage Models

Our primary usage model is a single system field deployment on a consumer laptop or desktop. In this model, the block hash database is stored locally or on a piece of removable media. The current storage capacity of commodity drives and external media is as large as a few terabytes in size. This is sufficient to store the block hashes of as many as one billion files. To store the MD5 block hashes of 1 billion files with an average size of 512 KB and a file block size of 4 KiB requires 2 TB of storage. This size will fit into the largest storage capacity of a commodity system or external storage device available for purchase today.

The limitation of using a single system model is the available memory. The current maximum memory available for a consumer laptop system is 16 GiB. As discussed in Chapter 7, the conventional DBMSs perform best when the database fits into memory. For the conventional DBMSs studied, 16 GiB can contain the block hash database for 100 million block hashes–the 1 billion hash database has an average size of 112 GB. For a block size of 4 KiB, 100 million block hashes represents 400 GB of content. This amount of content can support a few hundred million files with an average file size of 512 KB.

Another limitation is that only one examiner can use the system at a time, as the database is only locally available. However, this model is typical for field deployed systems and we focus our analysis on the single system model for this thesis.

A second usage model is the client/server model. In this model, the block hash database is stored on a remote server and accessed by many clients simultaneously. Similar to the single system model, the server can store the database locally or use external media. Servers typically have significantly more local storage space than a laptop or desktop. Servers also have more memory, typically a couple hundred gigabytes which easily supports the 1 billion hash database stored by all the conventional DBMSs used in this thesis.

The client reads the target media and sends the sector hashes to the database for comparison. For this model, we must consider the network throughput and latency. Throughput will limit the maximum number of hashes that can be searched per unit time. Latency is an issue with simplistic designs that do not rely on asynchronous Remote Procedure Calls (RPC).

The final usage model is a distributed database model. In this model, the hash database is split between multiple database servers. Because hash values are evenly distributed it is trivial to parallelize the database using prefix routing [8]. A cluster with 1,000 servers that each manage a database of 1 billion 4 KiB blocks can address four petabytes of known content. The benefit of the distributed database model is that it maintains the performance of a smaller database because each server only manages 1 billion hashes. Similar to the client/server model, the effects of network throughput is a factor but the impacts of latency are minimal if the lookups are batched and pipelined.

1.5 Chapter Outline

The following chapters discuss several of the design issues for implementing sector hashes for forensic triage analysis. Chapter 2 discusses prior work using content hashing to identify files. Chapter 3 provides a classification framework to discuss file block types. Chapters 4 and 5 discuss an experiment to determine the number of distinct blocks in three large file corpora of over 15 million files including user-generated documents, system files, legitimate and malicious software. Chapters 6 and 7 discuss an experiment to determine if conventional databases can meet the performance requirements of our file block hash database. Chapter 8 concludes, discusses the limitations of sector hashing and presents future work.

The major contributions of this thesis are: (1) the empirical evidence of distinct blocks in millions of files of various file types that can be used to identify a specific file, and (2) relative performance analysis of conventional DBMSs in storing 1 billion hashes.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Prior Work

This review spans two major forensic research areas; file identification and large hash databases.

2.1 File Identification

Traditionally, files are identified in forensic processing using file system metadata and carving. File metadata consists of information such as file name, creation time, size and the location of the file on disk. The metadata is stored in file system data structures that must be intact and decodable. This is the most straight-forward method to identify a file but metadata is trivially modified to hide the presence of a file without corrupting the contents.

File carving is the practice of searching an input for files or other kinds of objects based on content, rather than on metadata [9]. It is a powerful tool for recovering files and fragments of files when metadata is corrupt or missing either due to deleted files or damaged media. Most file carvers operate by looking for file headers and/or footers, distinct tags at the beginning and end of the file, and carving out the blocks between these two boundaries. More complex methods are needed to handle fragmented files. Various file carving methods are listed in Table 2.1.

Hash-based carving is the same idea as the distinct block identification presented in this thesis. *frag_find* is a forensic tool that performs hash-based carving [10]. The tool greedily searches the disk image for the longest run of sectors that match a contiguous series of file blocks. *frag_find* stores the entire sector hash database in RAM using an Standard Template Library (STL) map (a red-black tree). Its operation relies on the existence of distinct blocks and distinct block sequences. Although *frag_find* was published several years ago, there was no follow up work until this thesis and there has never before been a study of the prevalence of distinct blocks.

Dandass et al. present a case study where they analyzed hashes for over 528 million sectors extracted from over 433,000 files. They computed SHA-1, MD5, CRC32 and CRC64 hashes for each sector and compared the algorithms according to the number of false-positive indications and the storage capacity for the entire hash collection. The authors found no collisions with either the SHA-1 or MD5 algorithms but found that CRC64 had low collision rates and required only half the storage space. Dandass et al. conclude that CRC64 could be used as a filtering algorithm to extract sectors that do not match sectors from a collection of known illicit files [11].

Carving Method	Description
Block-based carving	Analyzes each block of the input to determine if the block is part of a file.
Statistical carving	Analyzes certain characteristics or statistics of the input, such as entropy, to determine which parts make up the file.
File structure carving	Carves files based on the internal structure of file types.
Semantic carving	Analyze the meaning of the input, such as linguistic analysis.
Carving with validation	Uses a file type validator to confirm carved files.
Fragment recovery carving	Reassembles two or more fragments to form the original file or object.
Hash-based carving	Hashes portions of the input and searches for matches to hashes of known files.

Table 2.1: Garfinkel and Metz propose the listed methods as a file carving taxonomy [9]. File carving tools that use file meta data or the file's internal structure are usually only effective at identifying fully intact and contiguously stored files. Tools based on the other methods, block-based, statistical, semantic, validation, fragment and hash-based, can identify fragments of a file by searching for exact matches or characteristics that are prevalent throughout a file.

However, follow up work by Garfinkel questioned this work as modern MD5 implementations are actually faster than CRC64 implementations and MD5 can take the same amount of storage as CRC64 if only half of the hash is retained.

The EnCase File Block Hash Map Analysis (FBHMA) EnScript is another sector hashing tool that searches for file blocks in disk sectors [12]. The script creates a database of target file block hashes from a master list and searches selected disk sectors for the file blocks. FBHMA also carves files using sector hashing, including files that have been partially overwritten or damaged. This tool was primarily designed to search file slack space, unused disk areas and unallocated clusters and not entire disks. It is not optimized for full media analysis and cannot perform sampling.

The *md5deep* tool suite also supports sector hashing [13]. *md5deep* is a set of tools to compute cryptographic message digests, including piecewise hashes, on an arbitrary number of files. The tool supports searching for file block hashes in media sectors. We used *md5deep* extensively in this thesis to compute the block hashes of our file corpora. Like the other sector hashing tools, *md5deep* is not optimized to support a large database of hashes and cannot perform sampling.

Wells et al. used block hash filtering to extract the most interesting data from a mobile phone for forensic investigations [14]. They divide media into small fixed-sized overlapping blocks and use block hashes for deduplication. The media blocks that match a library of known media block hashes computed from other phones are excluded under the assumption that matching blocks do not contain information of interest to investigators. According to their findings, their method reduces the amount of acquired data from collected phones by 69%, on average, without removing usable information.

2.2 Large Hash Databases

A critical issue with sector hashing is the performance requirements for a large and fast database of file block hashes. For the single system usage model, the database must be small enough to fit on a consumer laptop or external storage device and the query rate must be fast enough to allow for rapid triage, identifying potential files of interest as quickly as the media can be read. There has been several research initiatives to determine the best method to store and query large collections of sector hashes.

Collange et al. present a file fragment data carving method using Graphical Processing Units (GPUs). They compute hashes for every 512 B disk sector and compare the hashes to 512 B blocks from known image files. If there is a match, then the disk is flagged to signify that a file of interest potentially existed on the disk and that it requires deeper analysis. Taking advantage of the multiple cores in a GPU, they implement a parallel pattern matching engine that can process every 64 B fragment aligned on 32-bit boundaries in disks at a sustained rate of approximately 500 MB/s [15].

Farrell et al. evaluate the use of Bloom filters (BFs) to distribute the National Software Reference Library's (NSRL) Reference Data Set (RDS) [16]. The NSRL RDS is a collection of digital signatures of known, traceable software applications [17]. The evaluation was conducted with version 2.19 of the NSRL RDS that contains approximately 13 million SHA-1 hashes. Bloom filters were thought to be an attractive way for handling large hash sets because the data structure is space efficient. Farrell et al. could only obtain 17 K to 85 K RDS hash lookups per second using SleuthKit's hfind command and only 4 K lookups per second using a MySQL InnoDB database. Using a new BF implementation on the same hardware, query rates between 98 K and 2.2 million lookups per second were achieved. However, using BFs makes it dramatically easier for an attacker to construct a hash collision in comparison with a collision-resistant function such as SHA-1. The authors comment that an attacker could leverage the vulnerability to hide illicit data when BFs are used to eliminate "known goods" [16].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Taxonomy

This chapter presents a taxonomy for classifying blocks that we will use in the distinct block experiments.

3.1 Block Classification

The principal classification of a block is based on the number of times the block occurs in a corpus. Blocks that occur exactly once in the corpus are called *singletons*; blocks that occur exactly twice are called *pairs*; blocks that occur three or more times are called *common*. We created the three categories based on initial observations and then formed our hypothesis of the root causes for the frequency of occurrence to fit the observations.

Singleton blocks are those blocks that were found just once in the corpus. Pair blocks are those that were found twice. We hypothesize that pair blocks occur in files that are related, either because one file is embedded or contained in the other file or because one file is a modified version of the other file. Common blocks occur more frequently and we expect them to exist due to a commonality between all files or file types. For example, the block of all NULs (0x00) is a common block that is used to pad data in a file. In fact, the block of NULs is the most common block in our corpus.

The secondary classification of a block is based on the characteristics of its content. For our analysis, we classify blocks based on the byte entropy and the length of any repeating n-grams. We use Shannon entropy to measure the predictability of the byte values in each block [18]. An entropy score of 0 means that the block has 0 bits of entropy per byte and has a single byte value repeated throughout the block, for example the block of all NULs (0x00). An entropy score of 8 means that the block has 8 bits of entropy per byte and any byte value is equally likely to appear in the block, for example an encrypted block that resembles random data.

We also classify blocks based on the existence and length of the shortest repeating n-gram. An n-gram is a chunk of n-bytes. If a block consists of a byte pattern, or repeating n-gram, that repeats throughout the block, we call it a *repeating n-gram block*. The length of the n-gram can be as small as one byte and as large as half of the block size. For 512 B blocks, the largest repeating n-gram is 256 bytes and for 4 KiB blocks, the largest repeating n-gram is 2,048 bytes.

Classification	Definition
Principal Classification	
Singleton	Appear only once in a corpus
Pairs	Appear exactly twice in a corpus
Common	Appear three or more times in a corpus
Secondary Classification	
Entropy	Predictability of byte values in the block (0-8)
Repeating n-gram Block	Contains a repeating n-gram
N-gram Size	The size *n* of the repeating n-gram (0-half of block size)

Table 3.1: A list of the principal and secondary classification of file blocks in the corpora.

It is important to note that the last instance of the repeating n-gram may not fully repeat before the end of the block. For example, the string *abcabcab* consists of a repeating 3-gram ‘abc’ that is repeated twice and starts to repeat in the end of the string. It is also important to note that we do not look for blocks that consist of repeated byte sequences separated by variable data. For example, the string *abracadabra* obviously consists of a repeated 4-gram ‘abra’ but the pattern is interrupted with other variable data so the string is not considered a repeating n-gram.

We found many examples of common blocks with various entropy values and repeating n-grams. In general, we found that most blocks with low entropy or repeating n-grams were common, making these tests a useful prefilter.

Table 3.1 summarizes the block classifications used for our research.

3.2 Future Block Classification

An additional classification category that was considered but not used is based on printable ASCII strings found in a file block. Strings-based file identification is commonly used in forensics and malware detection and is also useful for our method [19,20]. All of the documents from the million government document corpus [21] (Govdocs1) were downloaded from government websites and most of the documents were in a human readable format (i.e. PDF, HTML, DOC, TXT). As a result, many of the blocks contain printable ASCII strings.

There are several expected types of string-based n-grams such as a string of all spaces, government related terms, any word in the English language and textual representation of numbers. If the strings are distinct we want to search for other blocks that have the same strings to identify other copies of the file. On the other hand, if the strings are common and repeated in many file we will not use it to identify a specific file.

Due to the time constraints of our research, we did not pursue string-based block classification

Classification	Definition
All spaces	Consists of a string of space characters (0x20)
Government related terms	Consists of government-related words
Textual Representation of #s	Consists of textual representation of numbers

Table 3.2: A list of potential string-based secondary block classification categories.

but believe that it should be studied in future research in distinct block identification.

Table 3.2 summarizes the string-based classifications.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Distinct Block Experiment

The objective of the distinct block experiment was to determine the occurrences of distinct blocks in a large collection of files. We hoped that a significant number of file blocks would be distinct in our corpus justifying the use of distinct blocks for identifying the presence of a single file. To this end we examined millions of real files and enumerated the number of distinct blocks.

To our knowledge, there are no previous studies analyzing the co-occurrence of blocks across such a large number of files and variety of file types. By using these corpora we can begin to make general conclusions about the true frequency of distinct blocks.

4.1 Resources

We used three existing file corpora to perform the experiments. Together, these corpora represent over 15 million files. The million government document corpus (Govdocs1) is a collection of nearly one million freely-redistributable files that were obtained by downloading content from web servers in the .gov domain [21]. The Offensive Computing Malware corpus (OCMalware) is a collection of approximately 3 million malware samples that were acquired from various collection and trading networks world wide [22]. The National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) Reference Data Set (RDS) is a collection of known, traceable software applications [17]. All three corpora consist of real data, and Govdocs1 and NSRL RDS contain additional provenance such as the original file name and creation date.

We used md5deep and Digital Forensics XML (DFXML) to compute and process the MD5 hash of each block in the file collection. md5deep is a set of cross-platform tools to compute message digests for an arbitrary number of files [13]. The tool can also compute piecewise hashes where files are broken into fixed sized blocks and hashed. md5deep can be configured to output the hashes in DFXML format. DFXML is an initial XML schema that provides common tags to allow for easy interoperability between different forensic tools [23].

We wrote tools to analyze the block hashes computed by md5deep using using Garfinkel's C++ DFXML processing libraries [23] as the Python implementation was too slow and required too

Count	Extension	Count	Extension	Count	Extension	Count	Extension
231,512	pdf	10,098	log	254	txt	14	wp
190,446	html	7,976	UNK	213	pptx	8	sys
108,943	jpg	5,422	eps	191	tmp	7	dll
83,080	text	4,103	png	163	docx	5	exported
79,278	doc	3,539	swf	101	ttf	5	exe
64,974	xls	1,565	pps	92	js	3	tif
49,148	ppt	991	kml	75	bmp	2	chp
41,237	xml	943	kmz	71	pub	1	squeak
34,739	gif	639	hlp	49	xbm	1	pst
21,737	ps	604	sql	44	xlsx	1	data
17,991	csv	474	dwf	34	jar		
13,627	gz	315	java	26	zip		

Table 4.1: The Govdocs1 corpus contains mostly human readable documents and images. Adobe PDF, Microsoft Office, HTML, log files and graphical image files make up the majority of the corpus. Through our block level analysis, we find that most files have correct extensions that match the file type, but some files do not. For example, the files with extension *txt* are all HTML documents.

much memory.

4.2 File Characteristics

Each of the three file corpora represent different types of files. OCMwalware and the NSRL RDS consist of mostly executable content and Govdocs1 consists of mostly non-executable documents. Based on the collection methodology and purpose of each corpus, OCMalware has mostly malicious content while Govdocs1 and NSRL RDS have mostly non-malicious files. It is useful that we analyzed corpora that have different types of files to determine if the existence of distinct blocks is a general characteristic. The following subsections provide additional information about the files in each corpus.

4.2.1 Govdocs1

The Govodcs1 corpus consists of 974,777 distinct files. The majority of the files are Adobe PDF, Microsoft Office, graphical image files and ASCII text. The average file size is 506 KB; 93% of files are larger than 4 KiB; 99% of files are larger than 512 B.

Table 4.1 summarizes the file types in the Govdocs1 corpus according to the file extension on the originally downloaded file. Although we do not confirm the file type for all files, we verified many of the file extensions during block content analysis discussed in Section 5.2 with the Unix *file* command and with visual inspection. There are instances of file extensions that do not match the file type, such as the files with the *txt* file extension that are HTML documents.

4.2.2 OCMalware

The OCMalware corpus consists of 2,999,076 distinct files. The average file size is 427 KB; 97% are larger than 4 KiB; 99.5% are larger than 512 B. The majority of the files are in the Windows Portable Executable (PE) format and include executables, and DLLs. Many of the files are packed with well-known software such as the Ultimate Packer for eXecutables (UPX) [24]. We do not provide a distribution of the file extensions because the corpus is currently organized using a directory structure such that each sample is named *malware.exe*. This naming convention was chosen to remind the user of the potential risk in working with the sample files.

We assume that the majority of the files are malicious based on the collection mechanism. However, only a few files were compared to publicly known virus signatures. These files were identified as malware by various antivirus software through VirusTotal.com [25]. It is an area of future research to confirm the purpose of each executable and classify each file according to its function.

4.2.3 NSRL RDS

The September 2009 NSRL RDS corpus consists of 12,236,979 distinct files. The average file size is 240 KB; 59% of the files are larger than 4 KiB; 90% of the files are larger than 512 B. Most of the corpus files are executable and some of the files may be considered malicious (i.e. steganography tools and hacking scripts [17]). The actual NSRL is not publicly available but NIST does publish the NSRL RDS. NIST provided the 4 KiB block hashes and file metadata used for this experiment.

4.3 Block Analysis Methodology

There were three phases carried out in the distinct block experiment: Computing Block Hashes, Classifying Blocks and Content Analysis of Select Blocks. The following sections describe each phase in detail.

4.3.1 Computing Block Hashes

We used md5deep to compute the MD5 cryptographic hash digest of every 512 B and 4 KiB block in Govdocs1 and OCMalware. NIST provided 4 KiB SHA-1 block hashes from their 2009 RDS. The 512 B block hashes were not available at the time of this research. At the completion of this phase, we had five sets of block hashes to use for analysis: Govdocs1 4 KiB, Govdocs1 512 B, OCMalware 4 KiB, OCMalware 512 B and NSRL 4 KiB.

4.3.2 Classifying Blocks

We created a C++ program *sector_stats* to analyze the hashes generated by md5deep. The program counts the number of *singleton*, *pair* and *common* blocks as defined in Chapter 3 for each of the five hash sets. We only include full blocks and ignored files and end blocks that were smaller than 512 B or 4 KiB in size.

Our program also checks for duplicate files using the MD5 file hash provided in the md5deep output and ignores any files that had been previously processed.

4.3.3 Content Analysis of Most Common and Random Blocks

We analyzed the top 50 most common blocks and 50 other randomly selected pair and common blocks in Govdocs1 and OCMalware to try to understand the reason that the blocks occurred in multiple files. For each block, we determine how many unique files and file types (as reported by the *file* command) the block occurred in as well as the entropy and pattern size of the block content. We also extract a sample of the block from a file in the corpus using *dd* and examine the contents with *hexdump*. When appropriate, we also viewed a sample of files that contained the block with *emacs* or the corresponding file viewer.

This analysis provided insight into the root cause of non-distinct blocks and identified several file-type specific block patterns that could be used to identify types of files.

Chapter 5 discusses the results of these experiments.

We did not perform this analysis with the NSRL blocks since we did not have a copy of the original files.

CHAPTER 5: Distinct Block Experiment Results

This chapter discusses the results of the distinct block experiment with the million government document corpus (Govdocs1), the Offensive Computing Malware corpus (OCMalware) and the National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) Reference Data Set (RDS) (NSRL2009).

5.1 Block Classification

As demonstrated in Table 5.1, the vast majority of blocks in each corpus are singletons and correspond to a single, specific file. This is not surprising. High entropy data approximates a random function. A truly random 512 B block contains 4,096 bits of entropy. There are thus $2^{4,096} \approx 10^{1,200}$ possible different blocks and they are all equally probable. It is therefore inconceivable that two randomly generated blocks would have the same content. The randomness of user-generated content is less than 8 bits per byte, of course, but even for content that has an entropy of 2 bits per byte there are still 1,024 bits of entropy in a 512B block, making it once again very unlikely that a block will be repeated by chance in two distinct files [7].

Table 5.2 shows that all kinds of user-generated content from Govdocs1, including word processing files and still photographs contains blocks that are only seen in one file in a large corpus of files. According to the distinct block hypothesis, these singletons can be treated as universally distinct blocks and used to find evidence that a particular file once existed on investigation media.

	Govdocs1		**OCMalware**		**NSRL2009**	
Total Unique Files	974,741		2,998,898		12,236,979	
Average File Size	506 KB		427 KB		240 KB	
Block Size: 512 B						
Singletons	911.4M	(98.93%)	1,063.1M	(88.69%)	n/a	n/a
Pairs	7.1M	(.77%)	75.5M	(6.30%)	n/a	n/a
Common	2.7M	(.29%)	60.0M	(5.01%)	n/a	n/a
Block Size: 4 KiB						
Singletons	117.2M	(99.46%)	143.8M	(89.51%)	567.0M	(96.00%)
Pairs	0.5M	(.44%)	9.3M	(5.79%)	16.4M	(2.79%)
Common	0.1M	(.11%)	7.6M	(4.71%)	7.1M	(1.21%)

Table 5.1: Occurrences of singleton, pair and common blocks in the Govdocs1, OCMalware and NSRL2009 corpora.

Extension	File	4 KiB Block	% Singleton	% Pair	% Common	Extension	File	4 KiB Block	% Singleton	% Pair	%Common
	Count	Count					Count	Count			
wp	15	393	100.00	0.00	0.00	*tif*	4	3	100.00	0.00	0.00
sys	2	7	100.00	0.00	0.00	*pst*	2	2	100.00	0.00	0.00
jar	16	18	100.00	0.00	0.00	*exe*	6	5	100.00	0.00	0.00
dwf	299	10551	100.00	0.00	0.00	*dll*	4	3	100.00	0.00	0.00
data	2	20	100.00	0.00	0.00	*chp*	3	8	100.00	0.00	0.00
sql	366	57060	99.98	0.02	0.00	*kmz*	692	68057	99.96	0.01	0.04
gz	13152	2165282	99.96	0.03	0.00	*docx*	161	8160	99.91	0.02	0.06
jpg	102287	9063011	99.86	0.05	0.09	*png*	3367	272818	99.75	0.18	0.07
pptx	212	140626	99.72	0.13	0.16	*text*	64539	12800405	99.61	0.23	0.16
gif	29552	721060	99.61	0.22	0.16	*squeak*	2	3169	99.46	0.13	0.41
kml	698	32707	99.46	0.41	0.13	*csv*	14414	831623	99.21	0.37	0.41
xml	36313	2018734	99.08	0.62	0.31	*xlsx*	45	1136	99.03	0.88	0.09
java	280	2624	99.01	0.84	0.15	*tmp*	121	3641	98.76	0.77	0.47
pdf	230703	32291471	98.74	0.36	0.89	*pub*	27	200	98.50	0.00	1.50
swf	3245	444247	98.28	1.32	0.40	*hlp*	148	880	97.73	2.27	0.00
bmp	71	7876	97.47	0.04	2.49	*xls*	63628	7095968	97.44	0.97	1.58
zip	26	254	97.24	1.57	1.18	*html*	173618	2725941	96.80	0.93	2.26
ppt	48952	30909344	96.63	1.70	1.67	*pps*	1560	898737	96.47	1.96	1.57
ttf	54	263	96.20	0.00	3.80	*log*	8990	1014152	95.27	0.51	4.22
doc	909817	7458713	95.23	1.39	3.38	*eps*	5410	771165	95.20	0.54	4.27
ps	909819	6920254	95.19	1.45	3.35	*js*	75	527	88.24	8.73	3.04
exported	6	50	76.00	24.00	0.00	*xbm*	17	166	68.67	31.33	0.00
txt	199	1804	43.79	13.41	42.79						

Table 5.2: The percentage of singleton, pair and common 4 KiB blocks for each file extension in the Govdocs1 corpus ordered by highest percentage of singleton blocks. Over 95% of blocks from most file extensions are singletons. The file extensions that have less than 95% singletons contain ASCII text and have a few nearly identical files, which results in a high percentage of non-distinct blocks. The shown file count only includes files that are larger than 4 KiB–files smaller than the block size are not included in the block hash database using our current methodology. The file counts are different from those shown in Table 4.1, which includes all files.

Table 5.2 also shows a few file extensions that have less than 95% singleton blocks. The files with extensions *js*, *exported*, *xbm* and *txt* have 88%, 76%, 69% and 44% singleton blocks, respectively. Using the *file* command and analyzing the content confirmed that the all of these files are HTML or ASCII text documents. The file data contents vary including logs, error messages, JavaScripts, bit map files coded as C code (the xbm format), and cascading style sheets. There are significantly fewer files with these extensions than other extensions. The high percentage of pair and common blocks come from a few cases of nearly identical files. Because there are such a small number of these files, the similar files have a larger effect on the overall percentages.

For example, the Govdocs1 files with extension *txt* have the lowest percentage of singleton blocks. These files are HTML documents that contain error messages about an unavailable resource. These files are not hosted *per se* on government web servers but are the server's response when a requested resource is unavailable. The Govdocs1 corpus was generated using a web crawler, and the crawler stores the server's response as a file in the corpus.

We found several sets of files that are nearly identical with similar size and content. The blocks in these files are mostly pair and common. There is only one line that differentiates these files that share 6 out of 7 4 KiB blocks, 85% of the file content. The first six blocks contain JavaScript that only appears in these eight files and in one other html document that does not contain an error message. The last block is HTML code that names the unavailable resource.

Analyzing the HTML headers and footers confirms that the files are all from the same government website. As discussed in Section 5.2.2, pair and low-occurrence common blocks (blocks that appear more than twice but not often) are typically found in files that are nearly identical or are different versions of each other. These blocks are not distinct according to our definition but can be used to identify distinct content, in this example the JavaScript that is exclusively used in an government agency's web pages.

There are also many singleton blocks within the standard operating system files represented in NSRL2009 that could be used for file identification.

The frequency of singleton blocks in the OCMalware corpus is lower than the frequency in Govdocs1 and NSRL2009 but still quite high. The number of pair and common blocks found in the malware samples is somewhat surprising since we know that polymorphic malware can encrypt the main body of the executable with a one-time key so that each copy has a different file signature but performs the same function. Encryption also changes the file block signatures. If the malware samples used this obfuscation technique, then we would find that the majority of the blocks were singletons and not repeated in any other file in the corpus, even if another file performed the identical function.

Fortunately, not all malware uses obfuscation and not all obfuscation techniques are highly sophisticated. As discussed in Section 5.2, some malware variants only differ in a few blocks throughout the file: This effects file-level signatures but would still allow block-level signatures to identify the file. Without knowing more detail about the function and obfuscation techniques of the samples in the malware corpus it is difficult to determine the root cause of the repeated content. However, our results show that although the majority of blocks only occur in one file in the malware corpus, some blocks are shared among various unique malware samples.

We also found a similar percentage of singletons for the 512 B and 4 KiB block sizes for Govdocs1 and OCMalware. Each 4 KiB singleton block consists of a combination of eight 512 B blocks that are either singleton, pair or common, as illustrated in Figure 5.1. Since the percentages of singletons for both block sizes are similar, we conclude that most of the 4 KiB singleton blocks are made up of eight singleton 512 B blocks and we do not loose granularity by choosing a larger block size. For the remainder of this chapter, we discuss the properties of the 4 KiB blocks from each corpus unless otherwise noted.

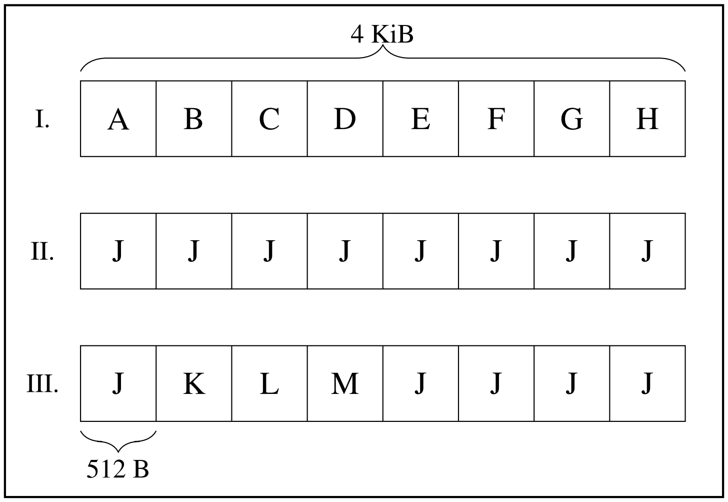


Figure 5.1: Within each singleton 4 KiB block there must exist eight 512 B blocks that are either singleton, pair or common within the corpus. The first example 4 KiB block is made up of eight singleton 512 B blocks, each of which only appear once in the corpus in this particular block. The second 4KiB block is made up of eight copies of a common 512 B block and the third 4 KiB block is made up of a combination of singleton and common 512 B blocks. Our results show that the majority of 4 KiB blocks in Govdocs1 are similar in construction to the first 4 KiB example block.

5.2 Block Content Analysis Results

We performed content analysis of 100 non-distinct blocks from Govdocs1 and OCMalware; the top 50 most common blocks and 50 randomly selected pair and common blocks in the corpus. We did not perform content analysis on blocks from NSRL2009 because we did not have the original files.

For each block, we examined the source files to confirm that the files were distinct and validated the file type using the *file* command. We also studied the characteristics of the repeating blocks to learn why the blocks were repeating and to begin developing general rules to determine if a sector contains a likely distinct block. An accurate rule will improve performance by allowing us to filter disk sectors read from the investigation media before making a database operation. But a rule that is too general, one that has a high false positive rate, will unnecessarily discard sectors that are distinct. Similarly, a rule that is too specific, one that has a high false negative rate, will unnecessarily pass sectors that are common, decreasing our processing rate.

We believe that these rules should be based on entropy and the existence of repeating n-grams; sectors below a certain entropy threshold and that have a repeating n-gram contain blocks that are likely not distinct. The block content analysis results confirm that highly common blocks have these properties. However, it is necessary to perform statistical analysis on singleton, pair and common blocks to determine actual rules that could be used to filter sectors read from target

Govdocs1 Block Ranking	Frequency	Number of Files	Number of File Types	Entropy	Pattern Size (Bytes)	Description
1	88,503	5,079	5	0	1	NUL block (0x00)
2	59,250	2,996	5	0	1	All 1s (0xFF)
3-22*	10,634	2,400	1	2.3	20	Adobe Acrobat XREF Table
23	6,437	23	1	1	2	TIFF structure
24-28, 30-34*	2,996	2,996	1	3.3	n/a	Compound Document SAT
29, 35-36, 40*	2,934	274	4	2	4	JFIF structure
37-39, 41-50*	2,996	2,996	1	3.3	n/a	Compound Document SAT

Table 5.3: The top 50 most common blocks in Govdocs1 contain constant blocks of NUL and all 1s as well as data structures for the Adobe PDF, Microsoft Office, JPEG and TIFF image file format. All of the blocks have low entropy and most contain a repeating n-gram. Low entropy blocks or blocks that contain a repeating n-gram are most likely not distinct. This table lists the frequency, number of containing files, number of containing file types, entropy, pattern size and description for each block. Rows that have a '*' next to the block id show average measurements.

media. This is an area of future research.

5.2.1 Most Common Blocks

Each of the top 50 common blocks occurred thousands of times throughout each corpus across many different distinct files. Some blocks occurred in files of the same type while others spanned across files of different types. Some blocks occurred exactly once in each file while others repeated several times throughout each file.

Govdocs1

Table 5.3 presents a summary of the top 50 common blocks in the Govdocs1 corpus. We found that the most common blocks in Govdocs1 contain software-generated content as opposed to human-generated content. This is expected due to the internal structure of many document file types.

The two most common blocks occur 88,503 and 59,250 times, respectively. Block 1 contains all zeros (0x00) and block 2 contains all ones (0xFF). Both blocks are repeated across many different file types. Most likely, the block contents represent 'filler' data that is generated by the software used to create the file. Clearly, any sector that is all zeros or ones is not distinct.

Blocks 3-22 occur on average 10,713 times in the corpus and consist of a repeating n-gram of 20 characters; *0000000000 65535 f* plus a newline character. The block occurs in Adobe PDF files and is a default entry in the PDF cross-reference (XREF) table. A PDF XREF table contains offsets to all of the objects in the PDF file and allows for quick lookup [26]. The first ten digits specify the object's offset into the PDF file and the second 5 digits specify the object's generation number. The final character indicates if the object is free, represented with 'f', or in use, represented with 'n'.

Block 23 occurred 6,437 times in the corpus and consisted of a repeating n-gram of 2 characters, *0xFF 00*. This block occurs in embedded Tagged Image File Format (TIFF) images in Encapsulated PostScript (EPS) files.

Repeating n-grams are clearly not distinct because they are too common. There are only 256 (2^8) different blocks that contain a repeating uni-gram. The probability that two randomly generated repeating uni-gram blocks match is 1 in 256, 0.3%. Similarly the probability for two 2-gram blocks is 1 in 65 K (2^{16}), the probability for 3-gram blocks is 1 in 16 M (2^{24}) and so on. The probability that two repeating 20-gram blocks match is 1 in 2^{160}, highly unlikely. However, we find that the Adobe default XREF entry, a 20-gram, is very common in Govdocs1. This block occurs more frequently then expected because it is a standard block generated by the Adobe software. There are many other common repeating n-gram blocks in Govdocs1 that are software generated and make up a portion of the file format internal structure. Our results support our proposed rule-of-thumb that if a block contains a repeating n-gram, it is likely not distinct.

Blocks 24-28, 30-34, 37-39 and 41-50 occur on average 2,996 times in the corpus in Microsoft Office documents. The blocks have low entropy but do not consist of a repeating n-gram. The blocks are from the Microsoft Compound Document File Format Sector Allocation Table (SAT). The SAT is an array of Sector IDs (SecIDs), a 4-byte value, that list the internal file sectors where user streams are stored in the document [27].

Each entry in the SAT lists the index of the next SecID in the chain or a special reserved value that provides meta-information about the chain. For example, a SAT stored in little-endian order that has *0x01 00 00 00* at index 0, *0x02 00 00 00* and index 1, *0x03 00 00 00* at index 2, and *0xFE FF FF FF* at index 3, indicates that a user stream is stored in sectors 0 - 3 and that 3 is the last sector in the chain (a SecID value of -2 indicates that the sector is the last in the chain). A hexdump of three sample blocks with different SecID arrays is shown in Figure 5.2.

Blocks 29, 35-36 and 40 occur on average 2,934 times in the corpus in embedded JPEG files in various container files. Each block consists of a repeating 4-gram pattern *0x28 A2 80 0A*. The first occurrence of each block is preceded by a JPEG File Interchange Format (JFIF) Header. The header began with a Start of Image (SOI) marker, *(0xFF D8)*, followed by the JFIF Application Use (APP0) marker, *(0xFF E0)* in the Microsoft Office, JPEG and Macromedia Flash files and *(0xFF EE)* in the Adobe Acrobat files [28, 29]. The JFIF header in the Microsoft Office, JPEG and Macromedia Flash files also contained the identifier *"JFIF"*, *(0x4A 46 49 46*

Block 26 - Sector Allocation Table(SAT)

```
00000000  01 00 00 00 02 00 00 00  03 00 00 00 04 00 00 00
00000010  05 00 00 00 06 00 00 00  07 00 00 00 08 00 00 00
00000020  09 00 00 00 0a 00 00 00  0b 00 00 00 0c 00 00 00
00000030  0d 00 00 00 0e 00 00 00  0f 00 00 00 10 00 00 00
00000040  11 00 00 00 12 00 00 00  13 00 00 00 14 00 00 00
```

Block 31 - SAT

```
00000000  01 02 00 00 02 02 00 00  03 02 00 00 04 02 00 00
00000010  05 02 00 00 06 02 00 00  07 02 00 00 08 02 00 00
00000020  09 02 00 00 0a 02 00 00  0b 02 00 00 0c 02 00 00
00000030  0d 02 00 00 0e 02 00 00  0f 02 00 00 10 02 00 00
00000040  11 02 00 00 12 02 00 00  13 02 00 00 14 02 00 00
```

Block 32 - SAT

```
00000000  81 02 00 00 82 02 00 00  83 02 00 00 84 02 00 00
00000010  85 02 00 00 86 02 00 00  87 02 00 00 88 02 00 00
00000020  89 02 00 00 8a 02 00 00  8b 02 00 00 8c 02 00 00
00000030  8d 02 00 00 8e 02 00 00  8f 02 00 00 90 02 00 00
00000040  91 02 00 00 92 02 00 00  93 02 00 00 94 02 00 00
```

Figure 5.2: Govdocs1 blocks 24-28, 30-34, 37-39 and 41-50 were found in the Compound Document File Format Sector Allocation Table (SAT) and contain an array of 4-byte Sector IDs (SecIDs) that list the sectors that user streams are stored in. This diagram shows the first 80 bytes of three sample blocks with different arrays of SecIDs.

JFIF Header in Microsoft Power Point files

```
00000000  ff d8 ff e0 00 10 4a 46  49 46 00 01 01 01 01 2c  |......JFIF.....,|
00000010  01 2c 00 00 ff db 00 43  00 08 06 06 07 06 05 08  |.,.....C........|
00000020  07 07 07 09 09 08 0a 0c  14 0d 0c 0b 0b 0c 19 12  |................|
00000030  13 0f 14 1d 1a 1f 1e 1d  1a 1c 1c 20 24 2e 27 20  |........... $.' |
00000040  22 2c 23 1c 1c 28 37 29  2c 30 31 34 34 34 1f 27  |‘‘,#..(7),01444.’|
```

JFIF Header in Adobe Acrobat files

```
00000000  ff d8 ff ee 00 0e 41 64  6f 62 65 00 64 00 00 00  |......Adobe.d...|
00000010  00 01 ff db 00 43 00 0e  0a 0b 0d 0b 09 0e 0d 0c  |.....C..........|
00000020  0d 10 0f 0e 11 16 24 17  16 14 14 16 2c 20 21 1a  |......$....., !.|
00000030  24 34 2e 37 36 33 2e 32  32 3a 41 53 46 3a 3d 4e  |$4.763.22:ASF:=N|
00000040  3e 32 32 48 62 49 4e 56  58 5d 5e 5d 38 45 66 6d  |>22HbINVX]^]8Efm|
```

Figure 5.3: Many of the Govdocs1 common blocks are in embedded JPEG images. This diagram shows the first 80 bytes of the JFIF header in the block proceeding the instance of Block 29 in two file types. The JFIF header in a Microsoft Power Point file starts with *(0xFF D8 FF E0)* and includes the string *"JFIF"*. Note that the ASCII character at position 0x40 in the Microsoft Power Point JFIF header is the straight double quote character (0x22). The JFIF header in an Adobe Acrobat file starts with *(0xFF D8 FF EE)*.

00) [30]. Figure 5.3 shows a hexdump of both headers.

OCMalware

The top 50 common blocks in OCMalware have various types of content. We did not reverse engineer any of the malware samples because it is outside of the scope for this thesis. We generally determined block functions by statically analyzing the content when appropriate and comparing the malware blocks to other blocks from known files in the NSRL2009–we computed the SHA-1 of the top 50 common 4 KiB malware blocks to compare to NSRL2009.

Similar to the common blocks of Govdocs1, there are many occurrences of the NUL block and the uni-gram block with all 1s, the first and fourth most common blocks in the corpus, respectively.

Block 2 occurs 741,084 times and consists of a repeating uni-gram, *0x90*. The majority of the files that have block 2 are PE32 Executables for Intel x86 machines and the block was not found in the NSRL2009. The hex value *0x90* is the Intel x86 No Operation (NOOP) instruction that has no effect on the machine context but advances the instruction pointer [31]. A sequence of NOOP instructions is often used to pad the area around a target instruction when the exact location of the instruction is unknown. If the execution flow reaches the NOOP sequence, then the instruction pointer will 'slide' to the target location. NOOP slides are often used in buffer overflows and similar exploits [32].

We suspect that block 2 is a NOOP slide due to the nature of the files in OCMalware, but it would require additional analysis of each containing file to confirm the block's function.

Block 3 occurs 7,022 times and consists of a repeating uni-gram, *0x2E*. All of the containing files are PE32 executables. However, there is no 1-byte Intel x86 instruction with that value. This block matches blocks in NSRL2009. All of the containing NSRL files have file extensions for image file formats (i.e. JPG, GIF). The hex value *0x2E 2E 2E* is the RGB value for dark gray. Block 3 is probably included in the data portion of the executable and may contain an embedded image.

Blocks 5-50 occur on average 114,999 times and have high entropy of 7.22. The blocks occur at the same offset in the containing files. For example, block 5 occurs at offset 4,096 in 125,025 unique files: each file has a unique MD5 hash and file size. A sample of the containing files are as reported by VirusTotal.com, a free virus, malware and URL online scanning service [25]. VirusTotal.com compares hashes to 40 antivirus signature sets and identifies matching malware instances. The sampled file hashes match different variants of the YahLover Worm, a low risk virus that enumerates system files and directories [33].

All 46 high entropy common blocks occur in the same 125,000 files. Each file is 255 KiB, on average and the first 60 4 KiB blocks occur in each file at the same offset. The blocks don't match any blocks in NSRL so we know that the blocks are specific to these malware samples. We suspect that all of these files are variants of the YahLover worm and that the files were

OCMalware Block Ranging	Frequency	Number of Files	Entropy	Pattern Size (Bytes)	Description
1	13,396,994	547,662	0	1	NUL block (0x00)
2	741,084	7,022	0	1	Repeating n-gram block (0x90)
3	218,134	1,330	0	1	Repeating n-gram block (0x2E)
4	133,492	4,662	0	1	All 1s (0xFF)
5-50*	114,999	114,999	7.22	N/A	Blocks from variants of the same malware sample

Table 5.4: The top 50 most common blocks in OCMalware include four repeating uni-grams and 46 high-entropy blocks. The repeating uni-grams are the NUL block, the block of all ones, a potential NOOP slide and data block that matches image files in the NSRL2009. The high-entropy blocks occur exclusively in variants of a malware sample. This block can help identify other variants of the same file. This table lists the frequency, number of containing files, entropy, pattern size and description for each block. Rows that have a '*' next to the block id show average values.

slightly modified by adding bytes to the end of the file. The modifications change the file hash which would prevent detection by antivirus scanners but not the block hashes as demonstrated by our findings.

This is an important finding because the common blocks are distinct to a specific malware sample and can be used to find other variants of the malware. Other malware samples may share blocks as a result of hand-patching existing malware and code reuse.

5.2.2 Random Pair and Common Blocks

We analyzed 50 random pair and common blocks in Govdocs1. The common blocks occur between three and five times in the corpus. We call these blocks *low-occurrence* blocks because they are repeated in the corpus but not often. The results of our analysis show that many of the low-occurrence blocks occur in files that are extremely similar and have many of their blocks in common. This occurs when files are copied and modified, created using the same template, or embedded into the same files. In the case of Govdocs1, they maybe a result of the acquisition methodology. These blocks are not distinct according to our definition but in some cases can be used to identify a class of content, similar to how the high entropy malware blocks identify other variants of the same malware.

Other low-occurrence blocks occur in files that only share a few blocks in common and the files are visually distinct. It is not clear why the block is repeated in this instance. It could occur due to a similarity in internal file structure or a small shared embedded file–for example, a shared embedded font. No general conclusion as to the root cause of these blocks was identified.

The following subsection describes the types of low-occurrence blocks seen in Govdocs1.

5.2.3 Findings

Analysis of the 50 random low-occurrence blocks in Govdocs1 is summarized in Table 5.5. The containing files as named in the Govdocs1 corpus are listed. The blocks occur in files of the same type or in the same file. Most of the blocks have high entropy although there are occurrences of low entropy blocks and one block has a repeating n-gram.

Blocks 51-90 occur in files that are nearly identical sharing almost all blocks in common except for the first and last block, files with revisions of the same content, files that use the same template and embedded files. All such files share the majority of their blocks in common at the same offset.

Nearly identical files have similar file sizes and all of the blocks are identical except for the trailing block. The blocks only occur in these files throughout the corpus and and there are instances of high and low entropy blocks. For example block 51 occurs in two visually identical PDF files that have 100% of their full blocks in common.

Files with revisions of the same content have file meta-data that indicates as such. We visually inspected the files to confirm that the contents were very similar. These blocks occur at the same offset, generally have high entropy and are only common to the containing files. For example, block 72 occurs in two Microsoft Power Point files that have the same creation date, author and title. The first file, 208098.ppt, is labeled as revision 155 and was last saved in September 2005 and the second file, 723019.ppt, is revision 164 and was last saved 22 days later. Both presentations have similar content and visual style; 96% of the blocks in 208098.ppt are pair blocks that only repeat in the 723019.ppt.

Files that use the same Microsoft Word or Power Point template share many blocks in common. These blocks have high entropy and occur at the same offset. For example, block 74 occurred in two Power Point Files that were different sizes, created and edited by different users, had different content but both use the *Focused senses design* template indicated in the file meta-data. The Power Point files share 76 blocks that only occur in these two files.

Files that share embedded content contain the same embedded file. For example, block 78 occurs in 161839.ppt and 235468.ppt, two Microsoft Power Point presentations with different content and style. However, both presentations have an embedded PNG image of a government agency logo. Block 78 is one of 20 pair blocks that make up the PNG image.

All of the similar blocks, except for the template blocks, can be used to identify similar files; files that have different variants of the same content. Blocks that contain Microsoft Templates can be automatically identified using the NSRL RDS.

Each of the similar files has a unique file hash that would prevent identifying the similarity using file hashing. However, Roussev's similarity digest would rate these files as having shared content [34].

Blocks 91-95 only occur in the same file. For example, block 91 occurs in a log file with ASCII text. The block occurs twice in the file and contains the same lines of log date. The repeated blocks have low entropy. These blocks are distinct according to our definition because the repeat block occurrence happens in the same file. Since the block is not repeated in any other file, it can be used to identify a specific file.

Blocks 96-100 occur in different files of the same type that have different sizes and meta data and appear visually distinct and unrelated. For example, block 100 is high entropy and appears in two PDF files at different offsets. The files are mostly text and contain different content. The first file, 322064.pdf, is 2 pages long and contains text and an image of a United States map. The second file, 134635.pdf, is 24 pages and contains all text. The files share 94 blocks in common, more than 55% for each file, and the blocks are contiguous in each file.

It is not clear why these documents share so many blocks in common. If the blocks are specific to PDF files, then we would expect to see more instances occur in other PDF files throughout the corpus. Perhaps we would find more instances with a larger corpus or perhaps the blocks identify a specific computer that was used to create the file.

5.3 Initial Statistical Analysis of Block Types

As illustrated in Table 5.6, the singleton blocks have the highest average entropy, the lowest percentage of repeating n-grams blocks and the highest average n-gram pattern size. The common blocks have the lowest average entropy, the largest percentage of repeating n-gram blocks and the lowest average n-gram pattern size. This is not surprising because we expect blocks with high entropy to appear less frequently in the corpus than blocks with low entropy. We also expect longer patterns to appear less frequently then shorter patterns because there are more bits of entropy in the former.

Based on our analysis of the most common and low-occurrence blocks in Govdocs1 and OC-

Malware, we believe that any sector that has low entropy or a repeating n-gram is likely not distinct. If most files contain distinct blocks, then filtering sectors with likely non-distinct blocks will allow us to identify files and minimize the number of database operations. Additional analysis is required to affirm this hypothesis and determine an appropriate entropy threshold.

Govdocs1 Block Ranking	Frequency	Entropy	N-gram Size (Bytes)	File Type	File Relation	Containing Files	
Blocks Repeated in Similar Files							
51	2	7.85		PDF	Nearly Identical	398395.pdf	635062.pdf
52	2	5.45		TEXT	Nearly Identical	103501.html	186235.text
53	4	7.35		PDF	Nearly Identical	883566.pdf 887380.pdf	883682.pdf 902600.pdf
54	2	7.92		PPT	Revisions	400527.ppt	816306.ppt
55	2	7.78		PPT	Revisions	737795.ppt	831305.ppt
56	2	7.88		PPT	Revisions	125596.ppt	344997.ppt
57	2	7.92		PPT	Revisions	716143.ppt	896918.ppt
58	2	7.63		PPT	Revisions	823851.ppt	823850.ppt
59	2	7.95		PPT	Revisions	081192.ppt	222426.ppt
60	2	6.64		PPT	Revisions	060050.ppt	733179.ppt
61	2	7.96		PPT	Revisions	265577.ppt	718449.ppt
62	2	7.92		PPT	Revisions	520238.ppt	520257.ppt
63	2	7.85		PPT	Revisions	759319.ppt	930569.ppt
64	2	7.60		PPT	Revisions	470989.ppt	720950.ppt
65	2	3.59		PPT	Revisions	153678.ppt	158011.ppt
66	2	7.49		PPT	Revisions	219463.ppt	499297.ppt
67	2	3.61		PPT	Revisions	148699.ppt	470007.ppt
68	2	7.95		PPT	Revisions	550550.ppt	729042.ppt
69	2	7.91		PPT	Revisions	223916.ppt	432439.ppt
70	2	7.93		PPT	Revisions	852904.ppt	877644.ppt
71	2	7.94		PPT	Revisions	328305.ppt	769590.ppt
72	2	5.53		PPT	Revisions	208098.ppt	723019.ppt
73	2	7.88		PPT	Revisions	251761.ppt	251763.ppt
74	2	7.84		PPT	Common Template	064978.ppt	064978.ppt
75	2	7.92		PPT	Common Template	113862.ppt	113862.ppt
76	4	7.89		WORD	Common Template	328908.doc 510693.doc	448441.doc 552838.doc
77	5	1.675	18	WORD	Common Template	128593.doc 260394.doc 944415.doc	134249.doc 779316.doc
78	2	7.95		PPT	Embedded Image	161839.ppt	235468.ppt
79	2	0.96		PS	Similar	265414.ps	310640.ps
80	2	7.95		WORD	Similar	497046.doc	541565.doc
81	2	0.52		WORD	Similar	042097.doc	215070.doc
82	2	7.95		PDF	Similar	153927.pdf	512812.pdf
83	2	7.95		PPT	Similar	514540.ppt	876566.ppt
84	2	7.91		PDF	Similar	785678.pdf	823177.html
85	2	3.58		PPT	Similar	582372.ppt	593079.ppt
86	2	1.43		PS	Similar	265414.ps	310640.ps
87	2	4.05		PS	Similar	144630.ps	742084.ps
88	2	5.42		HTML	Similar	564700.html	844361.html
89	2	5.06		XML	Similar	370547.xml	649858.xml
90	3	1.43		XLS	Similar	675701.xls 688704.xls	677169.xls
Blocks Repeated in the Same File							
91	3	1.76		WORD	Same file	051904.doc	
92	2	2.25		PPT	Same file	656483.ppt	
93	2	4.29		WORD	Same file	900384.doc	
94	2	4.62		TXT	Same file	108573.log	
95	2	2.94		PS	Same file	974311.ps	
Blocks Repeated in Dissimilar Files							
96	2	7.95		PDF	Dissimilar	036978.pdf	698843.pdf
97	2	7.96		PDF	Dissimilar	503976.pdf	895665.pdf
98	2	7.95		PDF	Dissimilar	302985.pdf	321983.pdf
99	2	7.93		PDF	Dissimilar	630279.pdf	927000.pdf
100	2	7.94		PDF	Dissimilar	134635.pdf	322064.pdf

Table 5.5: Fifty random low-occurrence blocks in Govdocs1 occurred in a few files in the corpus and no where else. Some of the containing files were similar, sharing a large percentage of pair or common blocks and containing similar content. Special types of similar files include nearly identical files, files with revisions of the same content, files that shared a common template and files that shared embedded content. Other blocks repeated in the same file and still other blocks occurred in dissimilar files where there was no perceived similarity in content or file metadata. It is unclear why blocks repeat in dissimilar files although our results show that this scenario does occur.

Measurement	**Singleton**	**Pair**	**Common**
Average entropy (per byte)	6.68	6.61	5.52
Standard deviation	1.92	0.34	2.70
Percentage of blocks that contain repeating n-grams	0.12	0.48	10.62
Avgerage n-gram size (bytes)	755	557	315
Standard deviation	33	56	170

Table 5.6: The entropy and n-gram statistics for singleton, pair and common blocks in Govdocs1. As expected the singleton blocks have the highest average entropy and the common blocks have the highest percentage of repeating n-gram blocks with the shortest average n-gram size.

CHAPTER 6:
Data Storage Experiment

Deploying sector hashing for full media analysis and sampling strongly requires a high performance database. The database must be able to store the file block hashes of every file that we have ever seen be fast enough to support triage at disk I/O speed. Previously published sector hashing methods do not scale for this number of files [10, 12]. It is critical therefore that our application supports both a large and fast database to leverage the intelligence from many previous investigations instead of just a few target files.

A database with 1 billion hashes will support a sufficient amount of content for many applications. One billion (10^9) 512 B block hashes can reference 1 billion 512 B file blocks, which is 512 GB of known content, which is large enough to store each of the corpora analyzed in this thesis. Similarly, 1 billion 4 KiB block hashes can reference 4 TB of known content, which is larger than the collection of known child pornography at the National Center for Missing and Exploited Children [35].

Today, we can read a 1 TB (10^{12}) hard drive in 200 minutes at maximum I/O transfer rate [7]. A 1 TB hard drive has 2 billion 512 B sectors and therefore a maximum sector reading rate of 150 K sectors/second. Assuming that hashing is free, the disk sector hashes are available to compare to the file block hash database at the same rate. Therefore the database must support a query rate of 150 K lookups per second for full media analysis.

With media sampling the required rates are not as high. As discussed in Chapter 1, sampling just 1 million randomly chosen disk sectors can identify 4 MB of content with 98.17% probability, provided that each of the 8,000 content blocks are distinct. A 7200 RPM hard drive can perform approximately 300 seeks per second. If pre-sorted it is possible to read 1 million randomly chosen sectors in 30 minutes on most systems today. Thus, for the random sampling application, a database query rate of a few thousand transactions per second is sufficient [7].

Our goal for the data storage experiment is to determine if we can achieve sufficient database query rates to support media sampling.

6.1 Purpose

The purpose of this experiment is to determine the relative performance of several general purpose SQL and NoSQL database management systems (DBMS) for storing and performing queries on a large file block hash database.

A general purpose DBMS is typically designed to meet many usage requirements including data accuracy and availability, resilience to errors and loss, simultaneous read and write access by multiple users and enterprise scalability [36–39]. The usage requirements for the hash database are not as broad. The database is essentially a key-value store where the key is a cryptographic hash of the file block and the value is a file identification number and byte offset where the block occurs. For the single-system field-deployed usage model, the database will rarely be updated, and run on a single laptop accessed by one or few connections. The most critical usage requirement is that the DBMS can perform fast queries for a database with one billion rows. To our knowledge, there are no previous benchmark tests that focuses on the specific usage requirements of full media analysis with sector hashing.

We used standard configuration options for each DBMS to measure the database query performance. No external pre-filtering was performed, such as using a bloom filter to distinguish between queries of *absent* and likely *present* hashes in the database. However, the results in Table 7.1 suggest that the DBMSs use their own pre-filtering since absent queries are performed at a faster rate then present queries.

6.2 Methodology

We tested the performance of three SQL DBMSs: MySQL, postgreSQL and SQLite, and one NoSQL DBMS: MongoDB. We used the latest software at the time of our study as shown in Table 6.2. For each DBMS, we built four databases that contain a single *table* of 1 million, 10 million, 100 million or 1 billion *rows*. We performed hash query tests on the databases to determine the performance of all DBMSs. Note that in the MongoDB DBMS, *tables* are called *collections* and *rows* are called *documents* [40]. We use the terms *table* and *row* generally for all DBMSs. Since each database only has one table, we use the terms *database* and *table* interchangeably.

The experiment is separated into several steps described in the following discussion.

First, we generated the file block hash data and save it in tab-delimited plain-text files. There

```
897316929176464ebc9ad085f31e7284    0    0
b026324c6904b2a9cb4b88d6d61c81d1    1    1
26ab0db90d72e28ad0ba1e22ee510510    2    2
6d7fce9fee471194aa8b5b6e47267f03    3    3
48a24b70a0b376535542b996af517398    4    4
1dcca23355272056f04fe8bf20edfce0    5    5
9ae0ea9e3c9c6e1b9b6252c8395efdc1    6    6
84bc3da1b3e33a18e8d5e1bdd7a18d7a    7    7
c30f7472766d25af1dc80b3ffc9a58c7    8    8
7c5aba41f53293b712fd86d08ed5b36e    9    9
```

Figure 6.1: The data shown is the first ten lines of a hash data file. We created 12 hash data files for each database. The first column in the file is an MD5 *hash*, the second is a *fileid* identification number and the third is a byte *offset*. The first line represents a block from file 0 at byte offset 0 (the start of the file) with a MD5 hash of *897316929176464ebc9ad085f31e7284*. The files were used to bulk load data into the databases, the fastest load method for each DBMS.

are twelve data files in total; one for each of the 1 million, 10 million and 100 million row databases and ten for the 1 billion row database – the billion row database is split into ten data files of 100 K rows, one of which is the data file for the 100 million row database. Each row in a data file corresponds to a row in the database and contains information for a file block; an MD5 hash, file identification number, and byte offset as shown in Figure 6.1.

We chose to pre-generate the data as opposed to generating it at load time because file bulk loading is the fastest way to populate databases and each DBMS supports data loading from a comma or tab-separated file. Additionally, pre-generating the data prevents work duplication when loading multiple instances of the same database stored in different DBMSs. Another method for avoiding duplication is to build the database in one DBMS, query the data and load it into all other DBMSs. Although this would prevent storing the data in an additional format, the method is not as efficient as loading from a file.

Next, we created the database and the schema defining the *columns* in each *row* and the *data type* of each *column*. There are three columns in each table, *hash*, *fileid* and *offset* that correspond to the file block data as previously described. The data schema is discussed in detail in Section 6.3. The database creation commands that define the schema are listed in Table 6.1.

All the DBMSs support a database index, a search data structure created from a column in the database that significantly improves performance of queries on data stored in the indexed column. Without an index, each query is compared to every row in the database. Using an index causes slower write times because the index structure is dynamically built when the data is inserted. Building the index after all the data is written to the database allows for both fast

MySQL InnoDB:

```
create table hashes (hash char(32), fileid integer, offset integer) engine=innodb
```

MySQL MyISAM:

```
create table hashes (hash char(32), fileid integer, offset integer) engine=innodb
```

PostgreSQL:

```
create table hashes (hash char(32), fileid integer, offset integer)
```

SQLite:

```
create table hashes (hash text, fileid int, offset int)
```

MongoDB:

```
d[hashes]
```

Table 6.1: These DBMS commands create a hash database with columns *hash*, *fileid* and *offset*, a MD5 hash, file identification number and byte offset, respectively. A detailed discussion of the database schema is provided in Section 6.3. Note that MongoDB does not take column names or types when creating the database. The column names are provided on data insert and the type is dynamically determined to best fit the data.

write times and query times. It is also better to build the index from a fully populated database to ensure that it is well structured and will provide optimal search performance [41]. Therefore, the database index is not created until after the data are loaded.

Third, we loaded the database by performing a bulk load from the data file as previously discussed. After the data is successfully loaded, we build the database index from the *hash* column. Since we query the database to find matching file block hashes from the disk sectors, the *hash* column is an obvious choice for the index.

Finally, we ran a series of timed database query tests. We created a Python software test tool that connects to a database and performs queries for *present* or *absent* values, hash values that exist or do not exist in the database, respectively. Python supports modules for each DBMS as shown in Table 6.2. We discuss the algorithm to determine *present* and *absent* hashes in Section 6.3.

During each test, queries for either *present* or *absent* hashes are performed exclusively for a fixed amount of time. We measured the number of completed queries and the time to complete each query. We chose to take measurements after each query to eliminate the time taken on the

Database		**Python3 Module**	
MySQL	5.5.22	pymysql	0.5
PostgreSQL	9.1.3	psycopg2	2.4.5
SQLite	3.7.7.1	sqlite3	2.6.0
MongoDB	2.0.4	pymongo	1.9b1

Table 6.2: We used the latest versions of database software and the corresponding Python3 module. We built the binaries for MySQL, PostgreSQL, MongoDB and pymysql. All other binaries were acquired from a Fedora 16 package repository.

other portions of the Python program. The total time is calculated as the sum of the individual query times. Although measuring this frequently adds to the overhead of the program, we found that the majority of the allotted query time was spent actually accessing the database. Due to our access method, the time measurements include the execution time for the Python database module and for network interaction except for SQLite, which does not use a client/server model.

When searching a target disk, we expect that most of the disk sectors will not match the distinct blocks in our hash database because we do not expect all of the known-files to exist on one disk. Therefore, we are especially interested in the query rate for *absent* values.

Caching is used by all DBMSs to store recently accessed portions of the database and index– either with an internal cache or the Operating System file system cache. We clear all caches before a run of the experiment to start with a *cold cache* and run each test long enough to achieve a steady query rate with a *hot cache*. A cold cache is empty and does not contain any portions of the database. Handling a new query with a cold cache requires reading from the disk to access the corresponding portion of the database and storing the portion in the cache for future access. A hot cache contains portions of the database and there is a higher probability that a new query corresponds to a portion of the database in the cache, preventing a costly disk access. We discuss in Sections 6.5 and 6.6 the configuration settings to clear the internal database cache and Operating System cache, respectively.

We performed the experiments on a Dell R510 server equipped with Dual Xeon E5620 2.4Ghz processors (16 core, 12 MiB cache, 128 GiB main memory). The server ran a 64 bit version of Fedora 16 with Linux kernel 3.4.2-1.fc16.x86_64. The databases were stored on a 21 TB physical RAID partition running XFS, a high-performance journaling file system for 64 bit architectures [42].

6.3 Database Design

We designed the file block hash database to support a corpus of known files. Every database entry, or row, is a 3-tuple that consists of a block MD5 digest (*hash*), a file identification number (*fileid*), and a byte offset (*offset*). The *fileid* and *offset* identify the source file and starting byte location of the block, respectively. The *hash* is the key, or index, for the database and is used to find rows during queries.

The *hash* is stored as a 32 character string equal to the 32 hexadecimal characters of the MD5 hash digest. Although we originally chose a string representation of the hash for ease of use, it was later realized that a binary representation would have been more efficient, reducing the field size by half. Due to the time constraints, we were unable to rebuild and retest the databases using the binary data type. Using the more compact data type will reduce the data size by 40% and the index size by 50% increasing performance, significantly. However, the relative performance of each DBMS should be the same with either text or binary keys.

As discussed in Section 4.2, we analyzed corpora that each consisted of millions of files. We chose to use a 32-bit integer for the *fileid*, which supports up to 2^{32} files, over 4 billion files. We also chose to use a 32-bit integer for the *offset*. For a block size of 512 B, the offset field can support files sizes up to 1.8 TB. This is much larger than the file sizes found in the corpora presented in Section 4.2.

Similar to the *hash* data type, the *fileid* and *offset* data types could have been optimized for a specific file corpus. It would be much more efficient to represent the *fileid* and *offset* together as one 32-bit integer. For example, a 22-bit *fileid* and 10-bit *offset* can represent over 4 million files that are 512 KiB and under. The average file size for each corpora presented in Section 4.2 were all under 512KiB. This optimization would reduce the data size by 10%, but would not allow for growth in the corpus, so we do not recommend it for a general solution.

For the experiment, we generated each row of the database using a simple equation. For row *i*, the *hash*, *fileid* and *offset* are equal to $MD5(str(i) + \text{`}\backslash n\text{'})$, *i* and *i*, respectively. The *hash* is computed by taking the MD5 digest of the string representation of *i* plus the newline character.

We used generated data instead of real data because it was simple to create an absent or present hash for our timed query experiments. For a database of N rows, a present hash is created by choosing a random number x between 0 and $(N-1)$ and generating the hash $MD5(str(x) + \text{`}\backslash n\text{'})$. An absent hash is created by choosing a random number x and generating the hash

Database Schema Terms	
Term	***Definition***
N	The number of rows in a database.
F_i	A unique identifier for the i^{th} file in the corpus.
o	A byte offset.
F_i^o	The fixed-size block at offset o in F_i.
$str(x)$	The string representation of x.
$MD5(y)$	The 32 hexadecimal digits of the MD5 digest of y.

Database Schema Description			
Column	***Size***	***Experimental Value***	***Actual Value***
fileid	32 bits	$n \in 0...(N-1)$	F_i
offset	32 bits	$n \in 0...(N-1)$	o
hash	256 bits	$str(MD5(str(n) + \text{`}\backslash n\text{'}))$	$MD5(F_i^o)$

Table 6.3: Each entry in the file block hash database is a 3-tuple (*hash*,*fileid*,*offset*). The *hash* is the 32 MD5 hexadecimal characters (256 bits) of a block that exists in a file represented by a unique *fileid* at a specific byte *offset*. We list the experiment values that were used generated for the experiment. The values differ from the listed actual values that will be used for a real block hash database. The summary of each value and its size is listed for a database of N rows.

$MD5(str(x) + \text{`}x\text{'})$. By definition, the former value is in the database while the latter is not. Furthermore, since hashes are evenly distributed, our generated data has a similar distribution to the block hashes of actual files.

Table 6.3 summarizes the entries for the generated hash database used during the experiment and for the real hash database that will be created from a corpus of known files.

6.4 DBMS Overview

MySQL is advertised as the world's most used open source Relational Databases Management System (RDBMS) [43]. The software uses a client/server model and supports several back-end data storage engines for each table in a database [44]. One of the advantages of using MySQL is the flexibility of choosing the storage engine and optimizing a table for a specific use [45].

We measured the performance of the InnoDB and MyISAM storage engines. InnoDB is the default storage engine for MySQL version 5.5.5 and later and is best suited for write heavy environments where data reliability is most critical. MyISAM is the default storage engine for MySQL versions earlier than 5.5.5 and has high performance for read heavy environments where the data are not frequently updated [44, 45].

Both InnoDB and MyISAM support table indexing and use a B-tree data structure to store table indexes which allows for searches, insertions and deletions in logarithmic time. Both storage engines provide backup mechanisms although the InnoDB data reliability is more robust. Both engines also support encrypted and compressed data storage and retrieval [44].

PostgreSQL is another leading open source object-relational database management system (OR-DBMS). It is has been actively developed for over 15 years and has a strong reputation for reliability, data integrity, and correctness [38]. The software uses a client/server model, supports a single storage engine and provides several index types: B-tree, Hash, GiST and GIN [41]. Each index type uses a different algorithm to compare queries against the table index. The default index type is B-tree. PostgreSQL also provides backup mechanisms and supports encrypted and compressed data storage and retrieval.

SQLite is an open-source, self-contained, serverless, zero-configuration transactional SQL database engine. Instead of using a client/server model, the software writes and reads directly from database files on disk but still allows for multiple access at one time [39]. SQLite is known for its simplicity and small code size (the library size can be less than 350KiB) and is often used in embedded devices where the database must work unattended and without human support [46]. Like the other databases, SQLite uses B-tree index types.

MongoDB is an open source NoSQL database written in C++ [37]. It uses a client/server model and a document based model where each row of data is stored as a document object, a set of dynamically typed key-value pairs. An advantage of using the document model is that related data are stored together, unlike in relational databases where data is separated into multiple tables and related, or joined, during a query [47]. Co-locating related data allows for fast query performance.

MongoDB can automatically distribute data between multiple servers in a clustered database. Although this feature is not useful for a laptop field-deployed usage model, it may be beneficial to consider for a lab environment where multiple servers are available for increased performance. MongoDB supports indexing and uses B-tree index types.

Finally, MongoDB automatically configures itself to optimize the use of system resources. This is a difference from MySQL and PostgreSQL, both of which require users to tune the performance of the database. One advantage of the MongoDB configuration is that it allows developers to concentrate on application development instead of performance tuning. One disadvantage is that it depends on the software to make appropriate decisions for system use.

Table 6.4 summarizes the features of each database.

6.5 Database Configuration

We configured the database software to use 64 GiB, half of the server's memory, for an internal data cache. We use all of the memory to allow space for the file system cache to store the recently accessed portions of the database file. The following sections present the configuration

DBMS	Data Model	Usage Model	Index Type	Required Amount of Performance Tuning
MySQL (InnoDB)	Relational	client/server	B-tree/Hash	High
MySQL (MyISAM)	Relational	client/server	B-tree	High
PostgreSQL	Relational	client/server	B-tree/Hash/GiST/GIN	High
SQLite	Relational	disk, embedded	B-tree	Low
MongoDB	Document Object	client/server	B-tree	None

Table 6.4: All of the DBMSs support similar features such as database indexing and file loading. There are differences in the Data Model, Usage Model Index Time and Required Amount of Performance Tuning. All the DBMSs use a traditional SQL relational model except for MongoDB that uses a document object model. All of the DBMSs use a client/server model except for SQLite, which directly accesses the database from disk. All of the DBMSs use a B-tree, except for InnoDB and PostgreSQL, which support other index structures. Finally MySQL and PostgreSQL all require extensive tuning to maximize performance, although the software is functional with the default settings. By design, SQLite requires very little performance tuning and MongoDB does not provide a mechanism to customize the software's use of resources.

settings for each database and the summary is shown in Table 6.5.

6.5.1 MySQL

InnoDB Settings

The InnoDB storage engine maintains a storage area called the buffer pool for caching data and indexes in memory [48]. The buffer pool is managed as a list, using a variation of the least recently used (LRU) algorithm. The size of the cache is controlled by the *innodb_buffer_pool_size* parameter and was set to use 64 GiB of memory, half of the server's RAM.

We also configured InnoDB to use multiple tablespaces by enabling the *innodb_file_per_table* parameter. By default InnoDB stores all of its tables and indexes in one data file. With the multiple tablespace configuration, each InnoDB table and associated index is stored in a separate file on disk [49]. This happens with MyISAM by default. We found this setting after having size issues with the shared data file while building the one billion row table. We reconfigured InnoDB to store each table in its own file and successfully built all tables indexed on the hash values. This also minimized interaction between different runs.

The InnoDB storage engine has a special feature that automatically builds an in-memory hash table on-demand for index values that are accessed frequently. This feature is controlled by the *innodb_adaptive_hash_index* parameter. The hash index is built with a prefix of the original B-tree index key by observing the search pattern. Every index in the B-tree index may not appear in the hash index. The hash table allows InnoDB to take advantage of large amounts of memory and will result in significant query rate improvements when the table fits almost entirely in main memory. This feature is enabled by default for the InnoDB storage engine and is not available for the MyISAM storage engine [50].

Database	**Parameter**	**Value**	**Description**
Internal Cache Settings			
MySQL (InnoDB)	*innodb_buffer_pool_size*	64 GiB	The internal cache size for the table and index.
MySQL (MyISAM)	*key_buffer_size*	64 GiB	The internal cache size for the index.
PostgreSQL	*shared_buffers*	64 GiB	The internal cache size for the table and index.
SQLite	*cache_size*	64 GiB	The internal cache size for the table and index.
MongoDB	*n/a*	*n/a*	*n/a*
Additional Settings			
MySQL (InnoDB)	*innodb_file_per_table*	*Enabled*	Creates a separate file for each table and index.
MySQL (InnoDB)	*innodb_adaptive_hash_index*	*Enabled*	Builds an in-memory hash table on-demand when appropriate.

Table 6.5: We configured the internal cache for MySQL, PostgreSQL and SQLite to 64 GiB, half of the server's available RAM. MongoDB solely relies on the OS's file system cache and does not use internal caching. We configured the MySQL InnoDB engine to create a separate file for each database's table and index, the default storage method for all other database implementations. Finally, we maintained the default setting for the InnoDB engine that allows for on-demand creation of an in-memory hash table index for databases that fit into resident memory.

MyISAM Settings

The MyISAM storage engine maintains a key cache to store frequently accessed index blocks. The key cache stores blocks in a list structure that follows a least recently used strategy (LRU). Unlike the InnoDB storage engine, MyISAM stores the table data and index in two separate files for each table. The key cache stores frequently accessed blocks of the index file and relies on the operating system file system cache to store frequently accessed blocks in the data file [51].

The MyISAM key cache size is controlled by the *key_buffer_size* parameter which was set to 64 GiB for the experiment.

6.5.2 PostgreSQL

PostgreSQL uses a shared memory buffer to cache 8KiB pages of the table and index. Each table and index are stored in a set of 1 GiB disk files [52]. The *shared_buffers* parameter determines the size of the the data cache and was set to 64 GiB for the experiment

6.5.3 SQLite

SQLite uses an in-memory cache to store recently accessed 1KiB pages of the database disk file. The database table and index are stored in one file and per our configuration, each data set was stored in a separate database and disk file. The *cache_size* parameter determines the size of the cache and was set to 64 GiB using the a PRAGMA statement to modify the operation of the SQLite library [53].

6.5.4 MongoDB

MongoDB relies solely on the operating system to handle caching. Since the database uses memory-mapped files for all disk I/O the OS virtual memory manager allocates memory to load pages, typically 4KiB, of the database table directly into memory. MongoDB does not implement a second internal cache [54].

The advantage to this implementation is that MongoDB automatically uses all available memory as necessary to maximize performance without any additional configuration. The amount of virtual memory can be limited using the Linux *ulimit* command. The disadvantage is that MongoDB can potentially starve other processes for RAM or cause extreme thrashing if the memory demands are higher than the available RAM. The MongoDB developers recommend ensuring that data will fit in available memory to avoid significant performance loss [55]. This is a concern for our application, due to the large number of hashes that we need to store and query.

6.6 Server Configuration

We configured the server for high shared memory usage. There are several kernel parameters that determine how much shared memory any one process can allocate in its virtual address space. MySQL and PostgreSQL use shared memory to store their buffer caches so we modified these parameters to allow for larger caches.

The *shmmax* parameter defines the maximum size in bytes of a single shared memory segment and the *shmall* parameter defines the total amount of shared memory pages that can be used system wide [56]. The default values allow at most 32 MiB of shared memory per process and no more than 8 GiB of shared memory for all processes. We modified the parameters to allow for one process to allocate a 70 GiB shared memory segment and to limit the system wide shared memory to 70 GiB. This setting was sufficient for the 64 GiB buffer caches that we configured each database to use.

We used the *sync* command to flush file system buffers to disk and wrote 3 to */proc/sys/vm/-drop_caches* to clear the file system cache between runs of the experiment.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Data Storage Experiment Results

The results from the data storage experiment are summarized in Table 7.1. For each database of 1 million, 10 million, 100 million and 1 billion rows, we show the DBMS database size and the query rates after the millionth query and after 1,200 seconds (20 minutes) from the start of an experiment run. We delayed the rate calculation to allow the cache to warm up and to reach a steady query rate. Although there are cases where a DBMS requires more time to heat up the cache, using a standard threshold across all DBMSs captures the time differences to reach a steady query rate. The query rates are presented in units of transactions per second (TPS) where a *transaction* is defined as a single query to the database for a *present* or *absent* hash. A production system might batch multiple queries in a single transaction.

There are several major observations based on the results. The first is that none of the DBMSs demonstrate the required performance for a 1 billion row database. As discussed in Chapter 6, sector hashing requires a query rate of a few thousand queries/second for media sampling and 150 K queries/second for full media analysis. Our results show that most DBMSs have a sufficient query rate for sampling using a hash database of 100 million rows or less. After the millionth transaction on a 100 million row database, MySQL InnoDB and MongoDB reach over 2 K transactions per second for present hashes and all other DBMSs except PostgreSQL reach 1.6 K queries per second or higher. For the 1 billion row database, the performance significantly drops and none of the DBMSs achieve more than 180 queries per second, an insufficient rate for our purposes.

The second observation is that major performance drops are due to the physical size of the database and the duration of the experiment. There are significant performance drops between the 10 and 100 million row databases and the 100 million and 1 billion row databases. The performance drops for 1 billion rows are expected because the database sizes are approaching the size of available memory – the average billion row database size is 112 GB. However, the 100 million row databases are all 17.5 GB or less and can fit into the internal DBMS cache and the Operating System file system cache.

Analyzing the query rate over time shows that the DBMSs do not reach a steady query rate at the end of the 100 million and 1 billion row database experiment runs. Steady query rates are

	Index and Data	TPS after 1M Queries		TPS after 1,200 seconds	
DBMS	Size (GB)	Present	Absent	Present	Absent
1 Million Rows					
MySQL (InnoDB)	0.11	2.23 K	2.84 K	2.23 K	2.79 K
MySQL (MyISAM)	0.11	2.64 K	2.95 K	2.64 K	2.95 K
PostgreSQL	0.16	3.42 K	3.56 K	3.41 K	3.56 K
SQLite	0.09	32.83 K	33.33 K	32.90 K	33.33 K
MongoDB	0.19	3.07 K	3.26 K	3.07 K	3.26 K
10 Million Rows					
MySQL (InnoDB)	1.12	2.18 K	2.35 K	2.18 K	2.35 K
MySQL (MyISAM)	1.05	2.48 K	2.50 K	2.50 K	2.49 K
PostgreSQL	1.54	3.36 K	4.04 K	3.33 K	3.95 K
SQLite	0.96	28.66 K	33.32 K	30.38 K	33.32 K
MongoDB	1.75	3.06 K	3.24 K	3.06 K	3.23 K
100 Million Rows					
MySQL (InnoDB)	11.11	2.00 K	2.30 K	1.51 K	1.69 K
MySQL (MyISAM)	10.44	–	1.60 K	0.09 K	1.10 K
PostgreSQL	15.00	–	–	0.05 K	0.36 K
SQLite	9.60	–	32.12 K	0.11 K	18.56 K
MongoDB	17.49	2.92 K	3.26 K	1.34 K	3.26 K
1 Billion Rows					
MySQL (InnoDB)	104.16	0.18 K	0.18 K	0.15 K	0.15 K
MySQL (MyISAM)	94.93	–	–	0.04 K	0.08 K
PostgreSQL	150.00	–	–	0.04 K	0.08 K
SQLite	97.00	–	–	0.05 K	0.09 K
MongoDB	115.57	–	–	0.04 K	0.13 K

Table 7.1: Total Transactions per Second (TPS) for highest query rate after 1 million queries and after 1,200 seconds (20 minutes). Dashes indicate that 1 million queries were not completed. The smaller databases that are 15 GB and below, easily fit into the server's memory and can reach peak query rates if given enough time. We see the drop in query rates from the 10 million row to 100 million row databases because the experiments were not run long enough for the database to load into the cache. The 100 million row databases can easily fit into the 128 GB of available RAM on the server. Similarly, there is a drastic drop in the query rates for the 1 billion row databases. All of those databases, except for PostgreSQL can fit into RAM but will take too long to load using random queries. We can achieve higher steady query rates faster if the data is preloaded. Only MySQL MyISAM and PostgreSQL currently support index preloading.

achieved with the smaller databases. For some DBMSs the performance at 1 million and 10 million rows is almost identical, i.e. the SQLite absent query rate for both databases is 33 K TPS.

In Section 7.2, we discuss how the query rate changes over time. Based on the analysis, it is possible that all of the DBMSs can achieve the required query rate for media sampling with a 1 billion row database provided that the database can fit into the available memory. The entire database is loaded into memory over time after so many queries or with preloading prior to querying. The memory requirements for the 1 billion row database is not practical for a field-deployed laptop that has 16 GiB of RAM. Therefore, a custom storage solution that has smaller memory requirements is required. It is also potentially useful to optimize the data set to fit into memory. However, the 1 billion row database would have to be reduced by almost 90% to fit into the memory of a consumer laptop.

The third observation is that SQLite has the highest performance for absent queries on the databases with 100 million rows or less. After one million transactions, SQLite achieves 32 K transactions per second when querying for absent hashes in the database with 100 million rows, a 3.6% performance decrease for managing 100 times more data. We believe that SQLite performs so well because it directly accesses the database on disk and does not go through a networking layer, which is necessary for the client/server model used by the other DBMSs. For the one billion row database, SQLite has the third lowest transaction rate. However, it is possible that given more time, SQLite could reach similar performance as is demonstrated with the smaller databases, given that the SQLite 1 billion row database is 97 GB and can fit into the server's available memory.

7.1 Data Size

Database size is critical to query performance. As data is queried from the database, parts of the disk file that store the database are loaded into the file system cache and the DBMS internal cache if available. If the database can fit into memory, meaning that the storage file is smaller than the available RAM, then we expect that the DBMS will eventually achieve similar performance to an in-memory database once the file has been fully loaded into the cache.

We observed the database sizes using the client program of each DBMS. The results are summarized in Table 7.1 and include the size of the data and index for each database. Some databases separate the data and index into separate files but both files are accessed to locate the data and read the contents.

PostgreSQL has the largest data size with an average of 0.15 GB per million rows and SQLite has the smallest data size of 0.09 GB per million rows. PostgreSQL achieves the second highest transaction rate for the two smallest databases but has the lowest transaction rate for the larger databases. The experiment did not explore the effects of data size on performance between different DBMSs but our results indicate that database size effects the performance of a single DBMS where larger databases are slower than the smaller databases.

The average size of the billion row database is 112 GB. It would require the full 128 GiB of the server's RAM to store the billion row table in memory for fast query rates. Running on a high-end consumer laptop with 16 GiB of RAM, the billion row table is too large to achieve in-memory type performance with the current data set representations. The current data sizes would require many more disk reads to compare disk sectors to our collection of known file block hashes.

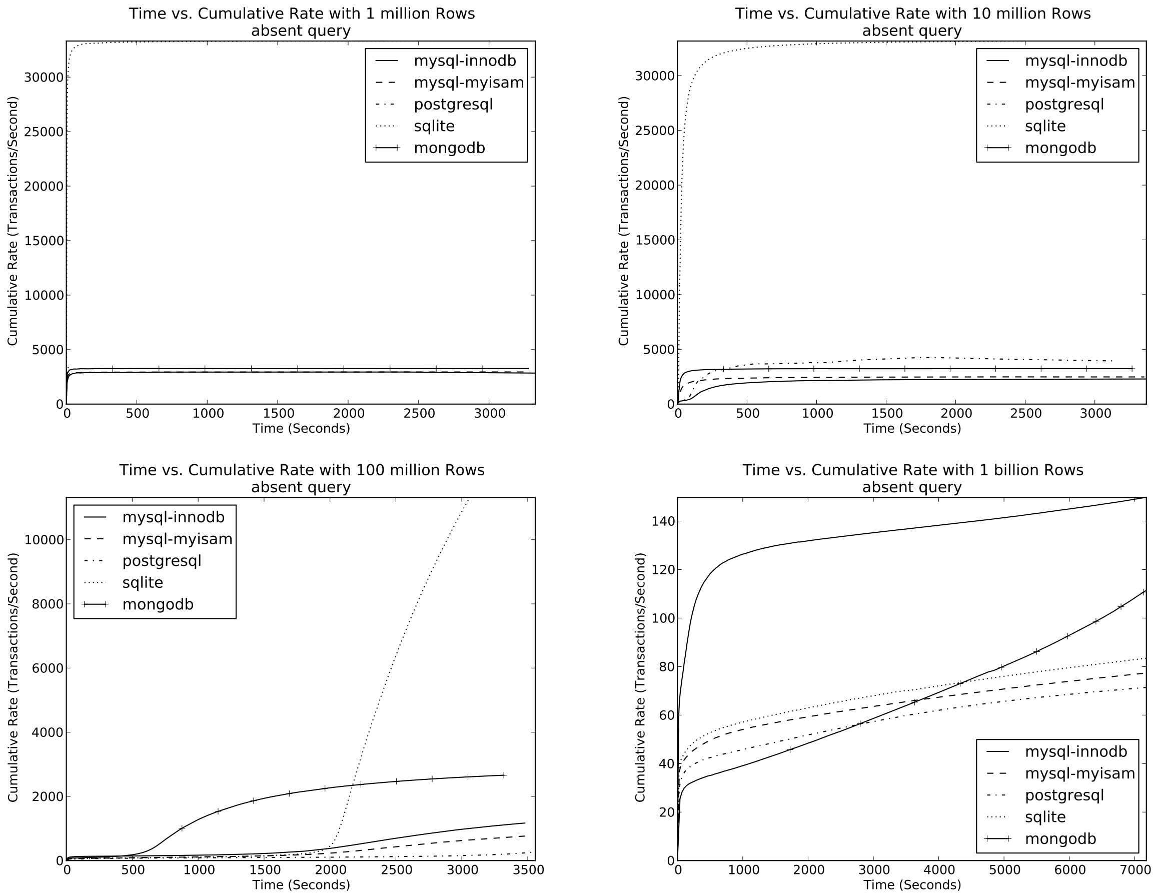
As previously mentioned, it is possible to make the database smaller using more compact data types to store the hash sets. Using one 32-bit value to represent a fileid and offset and using a 160-bit value to store the hash would significantly reduce the per-row data size. However, it is not clear if these optimizations would be sufficient to reduce the 1 billion row database below 16 GB. For future database analysis, we recommend optimizing the data set to minimize the size.

7.2 Query Rates

The graphs in Figure 7.1 clearly illustrate the effect of database size on the absent query rate. Although the present query rates were lower than the absent query rates, the relative performance and general characteristics of the graphs were similar. The query rate for 1 million rows hits the peak value almost immediately and is maintained for the duration of the test illustrated by the flat curves with zero slope. The average database size for 1 million rows is 0.13 GiB. The rate for 10 million rows doesn't hit the peak rate for a couple hundred seconds but then also maintains the rate. The average database size for 10 million rows is 1.1 GiB. As shown in the figure, there is a slight decrease in the steady rate for 10 million rows although there is ten times the amount of data being managed.

We see a dramatic decrease in query rates at 100 million rows. In Figure 7.2 it is clear that a "peak" rate is achieved by all DBMSs except for PostgreSQL. The curve for PostgreSQL still has a positive slope at the end of the experiment. The results listed in Table 7.1 do not count the first 1,200 seconds (20 minutes) or 1 million transactions towards the rate. This results in a higher measurement that is more accurate to the steady state query rate.

Clearly, a steady query rate has not been reached for the 1 billion row database experiments as shown in Figure 7.3. This is because the databases have not been fully loaded into the cache within the duration of the experiment. All the experiments were run for 3,600 seconds (1 hour) except for the 1 billion row database experiment that ran for 7,200 seconds (2 hours). The steady query rate is reached when the database has been fully loaded into the cache. This occurs after sufficient queries that access all parts of the database. Because we are querying for hashes of


Time vs. Cumulative Rate with 1 million Rows
absent query

Time vs. Cumulative Rate with 10 million Rows
absent query

Time vs. Cumulative Rate with 100 million Rows
absent query

Time vs. Cumulative Rate with 1 billion Rows
absent query

Figure 7.1: The query rate increases over time to a peak rate which occurs when the database is fully loaded into cache. After that time, a steady query rate is maintained for the duration of the experiment. For the 100 million and 1 billion row databases, the experiment was too Short for all DBMSs to fully load the database into the cache and a peak steady rate was not achieved. For the 100 million row and smaller database, SQLite has the best performance with 32 K TPS for absent queries. Higher rates are better.

randomly generated numbers the queries are well distributed and given enough time, the entire database will likely be accessed.

It is also possible to deliberately load the cache with index preloading. Currently, MySQL MyISAM and PostgreSQL support preloading. In MyISAM the index is loaded into the key cache with the *LOAD INDEX INTO CACHE IGNORE LEAVES* SQL command, the option to *IGNORE LEAVES* prevents loading non-leaf nodes from the index [57]. Index preloading is only supported with the MyISAM storage engine. In PostgreSQL, the functionality is available via a patch that provides the *pg_prewarm* function, although this patch has not yet been added to the main distribution [58].

Another method to preload the cache is to issue a database command that would access every part of the database. For example, if all hashes in the database are factored into one single calculation, then all hash values will be accessed. No testing was performed to confirm the effectiveness of this method. For future experiments, index hashing should be used to determine the peak data rate.

Even if the DBMSs can achieve sufficient query rates for 1 billion rows with a fully loaded cache, it is not a practical solution for the single-host, field-deployed system. Today, the maximum amount of memory available with a consumer laptop is 16 GiB. This is not sufficient to store the current 1 billion row database for any of the DBMSs. Due to the database sizes, the conventional DBMSs do not meet the performance requirements for full media analysis or sampling.

Several custom key-value storage solutions including hash-map, B-trees, red-black trees and sorted vectors were created and tested during a parallel effort [7]. The data structures were implemented using memory-mapped files and the NPS Bloom filter implementation [16]. Tests performed on a laptop with 8 GiB of RAM and 250 GiB Solid State Drives (SSD) attached via SATA and USB2, showed that the B-tree data structure had the highest performance for a 1 billion row database with 114.9 K TPS for absent queries and 3.7 K TPS for present queries after 1,200 seconds. Clearly, the custom key-value storage has higher performance than the conventional DBMSs and the solution is functional on a consumer laptop.

Based on the memory requirements and recorded query rates, the conventional DBMSs examined here are insufficient to support 1 billion block hashes for sector hashing with full media analysis or sampling.

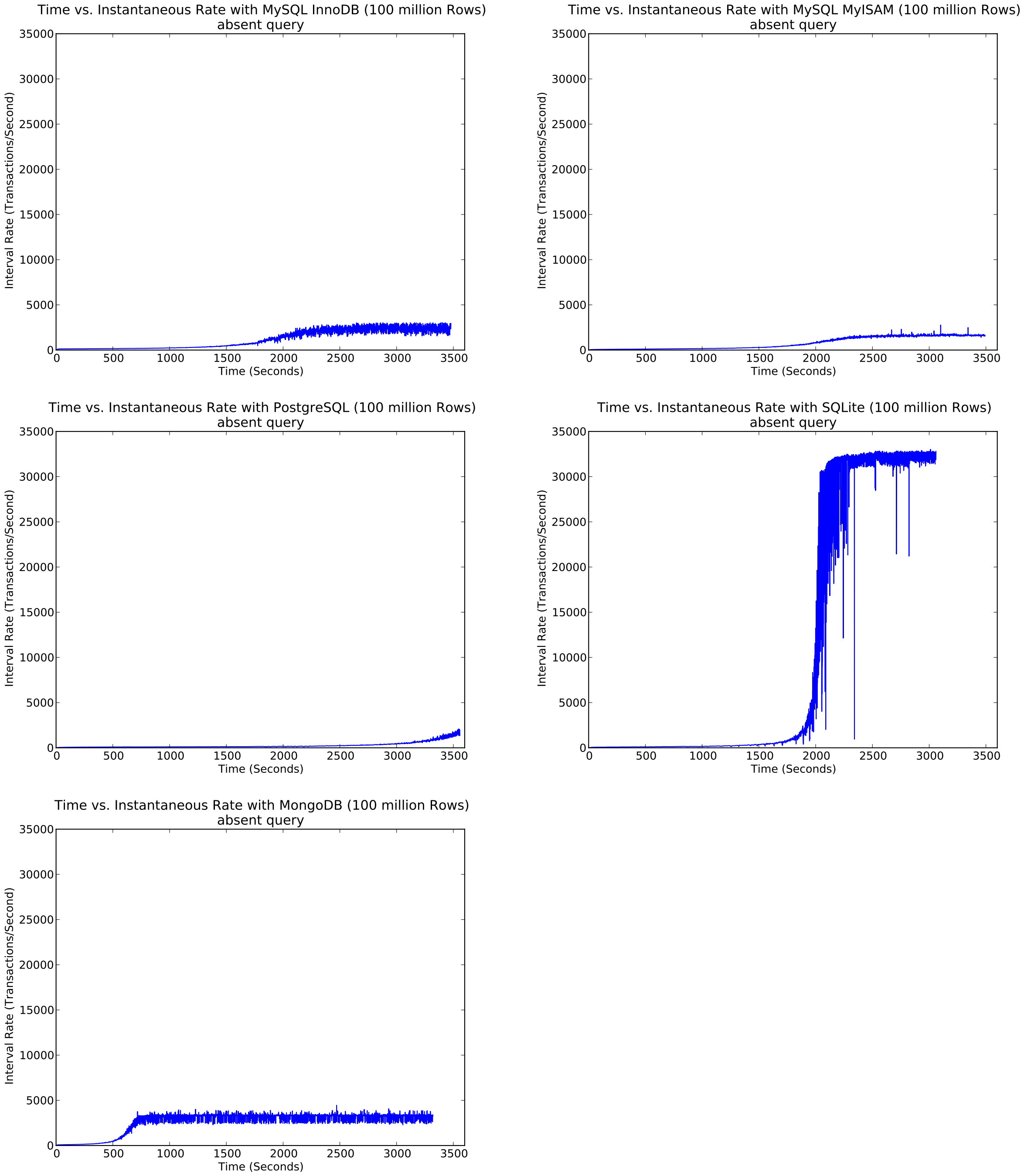


Figure 7.2: All the DBMSs except for PostgreSQL reach sufficient query speeds of more than 1 K TPS in 3600 seconds. As illustrated, PostgreSQL does not reach a steady query rate by the end of the experiment and the rate curve still has a positive slope. SQLite has the best rate at 32 K TPS. After each of the DBMSs reach a steady rate, there is still fluctuation perhaps due to other processes that are running on the system.

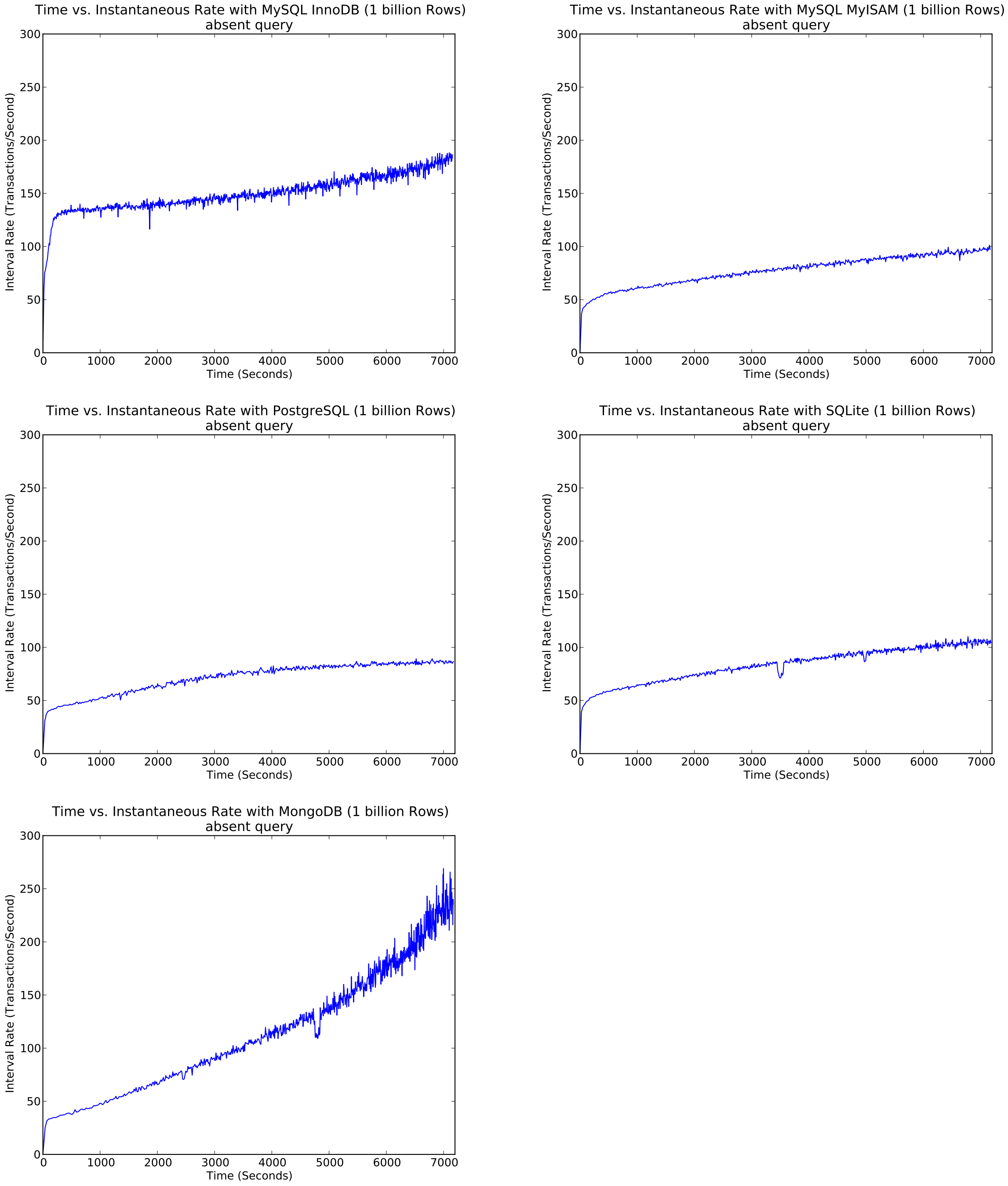


Figure 7.3: As illustrated, none of the DBMSs achieve more than 250 TPS for a billion rows. It is clear by the slope of the curve, that a steady rate has not yet been realized. It would take more time for the database to load into cache. All of the 1 billion row databases can fit into the server's 128 GiB of RAM except for PostgreSQL where the 1 billion row database is 150 GiB.

CHAPTER 8:
Conclusion

Forensic investigations often include massive amounts of digital data stored on various media. Given such volumes, examiners must be able to rapidly identify which media contains content of interest. Examiners identify content today by comparing files stored on the media to a database of known file hashes collected from previous investigations. Searching for content at the file-level is useful for in-depth analysis but not ideal for fast triage. Tools that parse the file system are slow and must support the version on the target media and tools that carve files based on headers and footers are not effective at identifying content from overwritten or partially destroyed files, or content that is fragmented into multiple locations on the media.

We present a forensic method that uses sector hashing to quickly identify content of interest. Using this method, we hash 512 B or 4 KiB disk sectors of the target media and compare those to a hash database of fixed-sized file fragments of the same size (which we call file "blocks"). Sector-level analysis is fast because we can parallelize the search process and sample a sufficient number of sectors to determine with high probability if a known file exists. Sector hashing is also file system agnostic and allows us to identify evidence that a file once existed even if it is not fully recoverable.

Identifying content with sector hashing depends on the existence of distinct file blocks, or blocks that only exist on media as a copy of the same original file. We analyzed three multi-million file corpora that contain real documents, system files, legitimate and malicious software and find that the overwhelming majority of the files contain distinct blocks that identify a specific file. We also found examples of non-distinct common blocks that can be used to identify content from files of a certain type and different versions of the same file. For example, we found distinct malware samples that appear to be variants of another malware sample in our corpus and share common blocks. This is an important finding because traditional file identification methods would require a hash of each sample and would not find similar variants.

Using sector hashing for distinct file identification also depends on the ability to store a large number of file block hashes that can be queried at a very fast rate. We tested the relative performance of several conventional SQL and NoSQL databases in managing a database of one billion hashes. Our results show that a custom storage solution is required for this method.

Related work by Young et al. [7], shows that a custom B-tree key-value store with a Bloom filter pre-filter is the best solution for a 1 billion row hash database. The combined B-tree and Bloom filter reaches 115 K TPS for absent queries and 4 K TPS for present queries, sufficient rates to support disk sampling for fast triage of large disks and full content analysis with only minimal performance degradation.

8.1 Limitations

There are several limitations to using sector hashes to identify target files. The main limitation is that the files must be sector aligned on the disk for successful identification. Since we generate the block hashes from the beginning of a file, if the file is not sector aligned on the target media, none of the containing sectors will match the file blocks in our database. One potential solution is to store multiple hashes for each block to account for non-sector aligned storage. This is unpractical due to the amount of storage required to maintain the additional hashes. Another is to abandon sector hashing and use Roussev's similarity digest [59] which overcomes this problem but with vastly increased processing times and storage requirements.

Fortunately, we expect most files to be sector aligned for performance reasons [7]. All three variants of the FAT file system (FAT12, 16 and 32), the *defacto* format for external storage devices, block align data. NTFS, the default file system for the current generation of Windows, block aligns files that are 1,024 B or larger. Any file smaller than 1,024 B is stored in the Master File Table (MFT) and not block aligned, but this is not a significant limitation as our method is not designed to work with such small files. Ext4, the default file system for most Linux distributions including newer versions of Android, block aligns data. ZFS, the most mature next generation file system, stores data in dynamically sized extents that consist of multiple sectors, and thus block aligns data [60]. The B-tree File System (BtrFS), the proposed future file system for Linux, stores data in dynamically sized extents consisting of multiple sectors. The extents are sector aligned. However, files smaller than 4 KiB are packed in the leaf nodes of the B-tree file system structure and not sector aligned [61], but again, this is of little consequence.

Generally, any file that is larger than the underlying disk sector size is sector aligned by the file system and can be identified using sector hashing. We cannot identify files that are smaller than the sector size but we expect that most files of interest are larger than the standard sector sizes. The overwhelming majority of files analyzed for this research are larger than 512 B and 4 KiB, the current standard sector sizes.

An adversary can maliciously attempt to modify files so that it is not stored on a sector boundary. For example, semantically NUL data can be added to the beginning of a file to change the file's alignment and ultimately create new block hashes for the same content. Due to the internal structure of most file types, it is not trivial to add bytes to the beginning of a file without changing how the file is rendered. It is much easier to add bytes to the end of a file which will change the file-level hash but maintains the block-level hashes.

It is easier for an adversary to change the file block hashes by making minor changes, or by encrypting the file. Similarly, an adversary can embed the file into another file. The containing file would be written on a sector boundary but the embedded file is aligned according to the file format. In either case, if the encrypted or packaged version of the file exists in the database and is transferred between systems without further modification, sector hashing can still identify the file blocks.

The last limitation is the ability to search for files stored on encrypted file systems. Bitlocker for NTFS and ReFS and FileVault2 for HFS+ encrypt data blocks as they are written to the storage medium and decrypt when it is read back. Because each drive is encrypted with a different key, the same data will be encrypted differently on different drives. Thus, sector hashing will not work with these drives unless the block device is read through the file system after the decryption key has been loaded or the drive is otherwise decrypted [7].

8.2 Future Work

There are many areas of future work within this research. It is worthwhile to explore string-based block classification to determine if a block is likely distinct. Strings are heavily used in forensics and malware detection and may be useful for sector-based forensics [19, 20].

Comparing block hashes across corpora is another useful research area. We compared the 4 KiB block hashes from NSRL2009 to the top 50 most common block hashes from OCMalware and only found matches between the NUL block and the blocks with repeating uni-grams (0xFF) and (0x2e). It would be useful to compare the full block set between OCMalware and NSRL2009 to determine if any of the malware samples share blocks with the software in the NSRL RDS. We expect to see some overlap between legitimate and malicious software with malware samples that are embedded in legitimate software.

Although the conventional DBMSs do not meet the performance requirements for 1 billion row databases stored on a consumer laptop, the DBMSs should still be considered for the client/server and the distributed usage models where larger amounts of memory are available. Some DBMSs, such as MongoDB, are designed for optimal performance in a distributed or clustered environment and automatically shard data between databases. Additionally, the DBMSs should be tested with a more optimized data set and using a solid state drive to determine if there are any performance enhancements based on these factors.

Sector hashing is a powerful tool for media forensics. Many files contain distinct blocks that can be used to identify a single file. Several tools currently use sector hashing but do not scale to support 1 billion block hashes [10, 12]. The biggest limitation in using sector hashing for full media analysis or sampling is managing the size and performance of the block hash database. But this is a technical challenge that can be readily solved today.

REFERENCES

[1] Guidance Software, Inc. EnCase Forensic, 2011. `http://www.guidancesoftware.com/forensic.htm`. Last accessed Dec. 3, 2011.

[2] Brian Carrier. The Sleuth Kit and Autopsy: Forensics tools for Linux and other Unixes, 2005. `http://www.sleuthkit.org/`.

[3] Golden G. Richard III and V. Roussev. Scalpel: A frugal, high performance file carver. In *Proceedings of the 2005 Digital Forensics Research Workshop*. DFRWS, New York, August 2005.

[4] Nick Mikus, Kris Kendall, and Jesse Kornblum. Foremost(1), January 2006. `http://foremost.sourceforge.net/`. Last accessed Dec. 3, 2011.

[5] Simson Garfinkel, Alex Nelson, Douglas White, and Vassil Roussev. Using purpose-built functions and block hashes to enable small block and sub-file forensics. In *Proc. of the Tenth Annual DFRWS Conference*. Elsevier, Portland, OR, 2010.

[6] Transition to advanced format 4K sector hard drives, May 2012. `http://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/`.

[7] Joel Young, Kristina Foster, Simson Garfinkel, and Kevin Fairbanks. Distinct sector hashes for target file detection. *Computer (to appear in IEEE)*, 2013.

[8] E.M. Bakker, J. Leeuwen, and R.B. Tan. Prefix routing schemes in dynamic networks. *Computer Networks and ISDN Systems*, 26(4):403–421, 1993.

[9] File carving. `http://www.forensicswiki.org/wiki/File_Carving`. Cited on September 2, 2012.

[10] Simson L. Garfinkel. Announcing frag_find: finding file fragments in disk images using sector hashing, March 2009. `http://tech.groups.yahoo.com/group/linux_forensics/message/3063`.

[11] Yoginder Singh Dandass, Nathan Joseph Necaise, and Sherry Reede Thomas. An empirical analysis of disk sector hashes for data carving. *Journal of Digital Forensic Practice*, 2:95–104, 2008.

[12] Simon Key. File Block Hash Map Analysis. The Computer Enterprise and Investigations Conference 2012, 2012. `http://www.ceicconference.com/agenda2012.htm`.

[13] Jesse Kornblum. md5deep and hashdeep—latest version 4.1, June 26 2011. `http://md5deep.sourceforge.net/`. Last accessed Feb. 18, 2012.

[14] Robert J. Wells, Erik Learned-Miller, and Brian Neil Levine. Forensic triage for mobile phones with DEC0DE. In *20th USENIX Security Symposium*, 2011.

[15] Sylvain Collange, Marc Daumas, Yoginder S. Dandass, and David Defour. Using graphics processors for parallelizing hash-based data carving. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, 2009. `http://hal.archives-ouvertes.fr/docs/00/35/09/62/PDF/ColDanDauDef09.pdf`. Last accessed Dec. 3, 2011.

[16] Paul Farrell, Simson Garfinkel, and Doug White. Practical applications of bloom filters to the NIST RDS and hard drive triage. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pp. 13–22. ACM, Anaheim, California, December 2008.

[17] National Institute of Standards and Technology. National software reference library reference data set, 2005. `http://www.nsrl.nist.gov/`. Last accessed 06 March 2009.

[18] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, July 1948.

[19] Computer Forensic Tool Testing Program. Forensic string searching tool requirements specification, January 24 2008. `http://www.cftt.nist.gov/ss-req-sc-draft-v1_0.pdf`, National Institute of Standards and Technology. Last accessed Dec. 3, 2011.

[20] Kent Griffin, Scott Schneider, Xin Hu, and Tzi cker Chiueh. Automatic generation of string signatures for malware detection. In *12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[21] Simson L. Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. In *Proceedings of the 9th Annual Digital Forensic Research Workshop (DFRWS)*. Elsevier, Quebec, CA, August 2009.

[22] Danny Quist. State of offensive computing, July 2012. `http://www.offensivecomputing.net/?q=node/1868`.

[23] Simson Garfinkel. Digital Forensics XML. *Digital Investigation*, 8:161–174, February 2012. Accepted for publication.

[24] Markus Oberhumer, Laszlo Molnar, and John Reiser. The ultimate packer for executables, 2009. `http://upx.sourceforge.net`.

[25] Cited on August 22, 2012.

[26] Kas Thomas. Portable document format: An introduction for programmers, 1999. `www.mactech.com/articles/mactech/Vol.15/15.09/PDFIntro/index.html`.

[27] Daniel Rentz. Microsoft Compound Document File Format, August 2007. `http://sc.openoffice.org/compdocfileformat.pdf`.

[28] JPEG file interchange format file format summary. `http://www.fileformat.info/format/jpeg/egff.htm`. Cited on August 20, 2012.

[29] Andrea Goethals. Action plan background: JFIF 1.01. December 2003. http://fclaweb.fcla.edu/uploads/Lydia%20Motyka/FDA_documentation/Action_Plans/jfif.pdf. Cited on August 20, 2012.

[30] JPEG file interchange format (JFIF). Technical Report ECMA TR/98, ECMA International, June 2009.

[31] Intel. *Intel ®64 and IA-32 Architectures Software Developer's Manual*, May 2012.

[32] Aleph One. Smashing the stack for fun and profit. *Phrack 49*, 7(49), November 1996.

[33] Virus profile: W32/YahLover.worm!o!06420B6749DE. http://home.mcafee.com/virusinfo/virusprofile.aspx?key=1290121. Cited on August 22, 2012.

[34] Vassil Roussev. An evaluation of forensic similarity hashes. volume 8, pp. S34–S41, 2011. http://dfrws.org/2011/proceedings/09-341.pdf.

[35] Simson L. Garfinkel. Personal communication. September 2012.

[36] MySQL Enterprise Edition. http://www.mysql.com/products/enterprise/. Cited on August 24, 2012.

[37] MongoDB. http://www.mongodb.org/. Cited on August 3, 2012.

[38] PostgreSQL about. http://www.postgresql.org/about/. Cited on August 6, 2012.

[39] About SQLite. http://www.sqlite.org/about.html. Cited on August 3, 2012.

[40] Collections. http://www.mongodb.org/display/DOCS/Collections. Cited on August 22, 2012.

[41] PostgreSQL index types. http://www.postgresql.org/docs/9.1/static/indexes-types.html. Cited on August 6, 2012.

[42] Xfs: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs/. Cited on September 2, 2012.

[43] Market share. http://www.mysql.com/why-mysql/marketshare/. Cited on July 29, 2012.

[44] Storage engines. http://dev.mysql.com/doc/refman/5.5/en/storage-engines.html. Cited on July 29, 2012.

[45] Mike Peters. MySQL storage engines, January 2008. http://www.softwareprojects.com/resources/programming/t-mysql-storage-engines-1470.html. Cited on July 29, 2012.

[46] Appropriate uses for SQLite. http://www.sqlite.org/whentouse.html. Cited on August 3, 2012.

[47] Philosophy, December 2011. http://www.mongodb.org/display/DOCS/Philosophy. Cited on August 3, 2012.

[48] The InnoDB buffer pool. http://dev.mysql.com/doc/refman/5.5/en/innodb-buffer-pool.html. Cited on July 29, 2012.

[49] Using per-table tablespaces. http://dev.mysql.com/doc/refman/5.0/en/innodb-multiple-tablespaces.html. Cited on July 30, 2012.

[50] Controlling adaptive hash indexing. http://dev.mysql.com/doc/refman/5.5/en/innodb-performance-adaptive_hash_index.html. Cited on July 30, 2012.

[51] The MyISAM key cache. http://dev.mysql.com/doc/refman/5.5/en/myisam-key-cache.html. Cited on July 30, 2012.

[52] Database page layout. http://www.postgresql.org/docs/9.1/static/storage-page-layout.html. Cited on August 8, 2012.

[53] PRAGMA statements. http://www.sqlite.org/pragma.html#pragma_cache_size. Cited on August 3, 2012.

[54] Caching, November 2011. http://www.mongodb.org/display/DOCS/Caching. Cited on August 8, 2012.

[55] January 2012. http://www.10gen.com/presentations/mongodb-los-angeles/2012/diagnostics-and-performance-tuning. Cited on August 2, 2012.

[56] Werner Puschitz. Tuning and optimizing Red Hat Enterprise Linux for Oracle 9i and 10g Databases. http://www.puschitz.com/TuningLinuxForOracle.shtml. Cited on August 5, 2012.

[57] Load index into cache syntax. http://dev.mysql.com/doc/refman/5.5/en/load-index.html. Cited on July 30, 2012.

[58] Raghavendra. Caching in PostgreSQL, April 2012. http://raghavt.blogspot.com/2012/04/caching-in-postgresql.html. Cited on September 2, 2012.

[59] Vassil Roussev. *Data Fingerprinting with Similarity Digests*, pp. 207–225. Springer, 2010.

[60] Kevin Fairbanks. An analysis of Ext4 for digital forensics. In *Proceedings of the 2012 DFRWS Conference*, 2012.

[61] Ohad Rodeh, Josef Bacik, and Chris Mason. BRTFS: The linux b-tree filesystem. Technical report, 2012. http://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/6e1c5b6a1b6edd9885257a38006b6130!OpenDocument&Highlight=0,btrfs.

THIS PAGE INTENTIONALLY LEFT BLANK

Referenced Authors

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia
2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California