NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SCALING BULK DATA ANALYSIS WITH MAPREDUCE

by

Timothy J. Andrzejewski

September 2017

Thesis Co-Advisors: Michael McCarrin
Marcus S. Stefanou

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2017	Master's Thesis 04-01-2013 to 09-22-2017

4. TITLE AND SUBTITLE

SCALING BULK DATA ANALYSIS WITH MAPREDUCE

5. FUNDING NUMBERS

6. AUTHOR(S)

Timothy J. Andrzejewski

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)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: NPS.2017.0021-AM01-EP5-A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Between 2005 and 2015, the world population grew by 11% while hard drive capacity grew by 95%. Increased demand for storage combined with decreasing costs presents challenges for digital forensic analysts working within tight time constraints. Advancements have been made to current tools to assist the analyst, but many require expensive specialized systems, knowledge and software. This thesis provides a method to address these challenges through distributed analysis of raw forensic images stored in a distributed file system using open-source software. We develop a proof-of-concept tool capable of counting unique bytes in a 116 TiB corpus of drives in 1 hour 41 minutes, demonstrating a peak throughput of 18.33 GiB/s on a 25-node Hadoop cluster. Furthermore, we demonstrate the ability to perform email address extraction on the corpus in 2 hours 5 minutes, for a throughput of 15.84 GiB/s, a result that compares favorably to traditional email address extraction methods, which we estimate to run with a throughput of approximately 91 MiB/s on a 24-core production server. Primary contributions to the forensic community are: 1) a distributed, scalable method to analyze large data sets in a practical timeframe, 2) a MapReduce program to count unique bytes of any forensic image, and 3) a MapReduce program capable of extracting 233 million email addresses from a 116 TiB corpus in just over two hours.

14. SUBJECT TERMS

hadoop, mapreduce, digital forensics, bulk data analysis, bulk_extractor, distributed digital forensics, data mining, big data

15. NUMBER OF PAGES 133

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

SCALING BULK DATA ANALYSIS WITH MAPREDUCE

Timothy J. Andrzejewski
Civilian, Department of the Navy
B.S., Georgia College & State University, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2017

Approved by: Michael McCarrin
Thesis Co-Advisor

Marcus S. Stefanou
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Between 2005 and 2015, the world population grew by 11% while hard drive capacity grew by 95%. Increased demand for storage combined with decreasing costs presents challenges for digital forensic analysts working within tight time constraints. Advancements have been made to current tools to assist the analyst, but many require expensive specialized systems, knowledge and software. This thesis provides a method to address these challenges through distributed analysis of raw forensic images stored in a distributed file system using open-source software. We develop a proof-of-concept tool capable of counting unique bytes in a 116 TiB corpus of drives in 1 hour 41 minutes, demonstrating a peak throughput of 18.33 GiB/s on a 25-node Hadoop cluster. Furthermore, we demonstrate the ability to perform email address extraction on the corpus in 2 hours 5 minutes, for a throughput of 15.84 GiB/s, a result that compares favorably to traditional email address extraction methods, which we estimate to run with a throughput of approximately 91 MiB/s on a 24-core production server. Primary contributions to the forensic community are: 1) a distributed, scalable method to analyze large data sets in a practical timeframe, 2) a MapReduce program to count unique bytes of any forensic image, and 3) a MapReduce program capable of extracting 233 million email addresses from a 116 TiB corpus in just over two hours.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

CART	Computer Analysis and Response Team
DELV	Distributed Environment for Large-Scale Investigations
DF	Digital Forensics
DDF	Distributed Digital Forensics
DFaaS	Digital Forensics as a Service
DFRWS	Digital Forensic Research Workshop
DoD	Department of Defense
EB	Exabyte
EWF	Expert Witness File
FNMOC	Fleet Numerical Meteorology and Oceanography Center
FTK	Forensic Tool Kit
GFS	Google File System
GiB	Gibibyte
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
IRS	Internal Revenue Service
MPI	Message Passing Interface
MR	MapReduce
MMR	MPI MapReduce

MiB	Mebibyte
NFI	Netherlands Forensic Institute
NPS	Naval Postgraduate School
PiB	Pebibyte
RAID	Redundant Array of Independent Disks
TiB	Tebibyte
TF-IDF	Term Frequency-Inverse Document Frequency
TSK	The SleuthKit
USN	U.S. Navy
UDP	User Datagram Protocol
USG	United States government
XIRAF	XML Information Retrieval Approach to Digital Forensics
YARN	Yet Another Resource Negotiator

Acknowledgments

First, I would like to express my gratitude to both my advisors, Michael McCarrin and Dr. Marcus Stefanou. Michael's desire and curiosity to explore new avenues in the field of digital forensics was truly infectious throughout this research. I would like to thank Dr. Marcus Stefanou for his willingness to join this research late and continued support to ensure I remained focused. I would also like to thank Dr. Mark Gondree for his early work in the project as well as his guidance on many of the technical issues encountered. I could not have completed this research without the guidance and input from each of these people, and for that I am truly grateful for their time and knowledge.

I would also like to thank Fleet Numerical Meteorology and Oceanography Center (FNMOC) for the opportunity to advance my education while working full-time in the N63 division. I thank my N63 colleagues and FNMOC leadership, past and present, for their support and flexibility while I worked an odd schedule to complete this research and degree.

I would like to thank my friends, James and Emmy, back home in Atlanta for their continued friendship and support as well as the many conference calls we had. The calls were a welcomed distraction filled with many laughs throughout this research process and degree.

Lastly, I would like to thank my whole family, immediate and extended, for their support and dedication in pushing me to achieve success in my life. I especially would like to thank my mom for her continued support and belief that I can achieve anything. I would also like to thank my dad for his continued support in keeping me focused on the goal even when it becomes tough to. A special thank you to my aunts and uncles for their support and encouragement to ensure I find a good work-life balance. My brothers and sister, I thank you for your love and understanding of my ways even though we are each at very different points in our lives. To my cousins spread out across the country, thank you for your continued motivation to push myself to achieve more.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Forensic analysts are faced with an increasing variety, quantity and complexity of data which they must analyze in a limited time. Information critical to investigations, moreover, is no longer limited to the standard personal computer. Today, we see users with multiple devices such as laptops, desktops, tablets and cell phones, each with increasing storage capacity. In addition to the devices a typical user possesses already, there are also many devices a user comes in contact with that may contain information crucial to a forensic analyst. Wearables, external hard drives, smart home devices and cloud-based services all add to the volume and variety of data that must be examined.

1.1 Motivation

The primary motivations for this thesis are to reduce the time digital forensic analysts devote to executing tools and to reduce the cost of a forensic investigation. Both are closely related, since reducing time to execute tools directly reduces overall cost. However, costs also include the tools themselves, as well as the time required to learn those tools and the hardware required to execute them. Any improvement in these areas will help bring digital forensic tools and capabilities to agencies and companies of a greater variety of budgets and manpower sizes. Additionally, our research is motivated by the need to bring simplicity to DF tools and allow forensic analysts to be experts at forensic examination instead of experts at understanding the tools.

Previous efforts to address these motivations resulted in DF tools becoming significantly more complex and parallelized on a single system while becoming costly to execute. Efforts to achieve higher throughput have focused primarily on increased parallelization on standalone, multicore systems. This trend is nearing its limits. In 2004, Roussev and Richard described in detail the need for distributed digital forensics [1]; then six years later Garfinkel reiterated "the coming digital forensics crisis" of growing storage size and insufficient time to analyze this data [2]. As late as 2016 the SANS Institute wrote that there have been "few efforts to discuss managing the increased volume, variety and velocity of incoming data as a big data issue" by way of motivating their proof of concept tool using Apache

Spark to extract strings of significance [3]. During the same time, especially in commercial industry, we have seen a dramatic increase of scalable cluster based algorithms designed for the purpose of handling growing volumes of data.

With these motivating factors, this thesis aims to address the following research questions:

1. Can the MapReduce paradigm be leveraged to provide a distributed computing method to reduce digital forensic tool execution time and cost?
2. What best practices should be used to implement a MapReduce approach to digital forensics?
3. Is the MapReduce solution to digital forensics scalable enough to keep up with growing volumes of data?

1.2 Relevance and Contributions

The work in this thesis is relevant primarily to the digital forensic community which is currently facing a data volume challenge: the amount of data available surpasses our capability to analyze it. This thesis provides a new capability to the forensic community by using a parallel processing method that has already demonstrated performance benefits in processing of textual data. We develop tools to bring those benefits to forensic tasks dealing with raw binary disk images.

We provide an InputFormat class that facilitates the analysis of raw binary images using MapReduce for parallelization. We also provide two MapReduce programs that illustrate the benefits of using MapReduce to tackle the growing data volume challenge. Finally, we perform timing measurements for data ingest, analyzing 116TiB of data to count byte values and extraction of significant email addresses.

Results from this work can help government agencies (federal, state or local) evaluate the benefits of a Hadoop cluster at their site. These results may be used further in corporate law offices or the courts by allowing them to complete a full analysis of all devices many of which may contain over several TiBs of storage capacity. As cloud computing continues to grow tools developed in this research may become more critical if the need to perform full analysis arises.

Digital forensics is a key tool used by the military to investigate cyber security incidents and to quickly process digital media and devices acquired from adversaries in the course of operations. Specifically the Navy benefits from intelligence acquired from digital media as a result of this research. Providing this intelligence quickly could mean a go or no go decision for an operation depending on the intelligence acquired. In addition to gathering intelligence the Navy benefits from enhanced ability to examine attacks against their own information systems.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides background information and terminology on topics required for understanding our research, including concepts in digital forensics, Big Data, Hadoop MapReduce and High Performance Computing. It also provides specifics regarding the use of MapReduce for our research. Chapter 3 covers previous work in digital forensic tools, Hadoop storage and processing of binary images, distributed digital forensics tools and data mining in digital forensics. Our Methodology and Results will be covered in Chapter 4 and Chapter 5. In Chapter 6, we lay out our conclusions and suggest future work that remains.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter aims to define the foundational concepts that underpin our research. We begin by introducing digital forensics (DF) and its history and present a sketch of the progression of digital forensic tools leading up to our work. In this section, we describe the bulk data analysis approach and explain why we chose to extend it. Next, we review some of the concepts and definitions of Big Data and describe how these relate to our goal of processing 116TiB. We then move to tools and methods to process large data sets of digital evidence, first defining MapReduce, a Big Data paradigm that allows processing of hundreds to thousands of tebibytes (TiB) or pebibyte (PiB). Finally, we define and compare two common distributed computing frameworks, High Performance Computing (HPC) and Hadoop MapReduce. Our aim is to move digital evidence processing into a distributed environment; therefore, it is important to understand these two common approaches. An understanding of each of these topics builds the foundations that are needed to understand our research methods and results.

2.1 Binary Unit Symbols

This thesis uses the International Electro-technical Commission (IEC) standard to represent sizes of files and storage systems. This standard is different than the International System of Unites (SI) because it uses base 2 instead of base 10. The IEC standard measures a gibibyte (GiB) to be 2^{30} whereas the SI standard measures a gigabyte (GB) to be 10^9 though the two are seen to be used interchangeably. The different values between the two standards is subtle with smaller units such as KiB and MiB, but as units increase this difference grows exponentially and may no longer be inferred from context [4]. For example, the exact difference between using GiB versus GB is 73,741,824 bytes.

2.2 Digital Forensics: Definition and History

Before its infancy (in what Mark Pollitt labels as "pre-history" or pre-1985) the closest thing to digital forensics, as it is defined today, were early system audits—reviews of system usage, efficiency and accuracy of data processing to detect fraud [5]. In the next several paragraphs

we review in detail digital forensics history (summarized in Figure 2.1). Pollitt describes this period as dominated by "*ad hoc*" volunteer based individuals [5] since no dedicated organizations existed. Investigations during this period were experiments with system administration tools, which as both Pollitt and Garfinkel point out, are best represented in Cliff Stoll's, *The Cuckoo's Egg* [2], [5] [6]. Before the Computer Fraud and Abuse Act of 1984 made computer hacking a crime, there was a greatly reduced desire to perform audits or DF investigations since corporations with main-frames did not have clear legal grounds to prosecute.

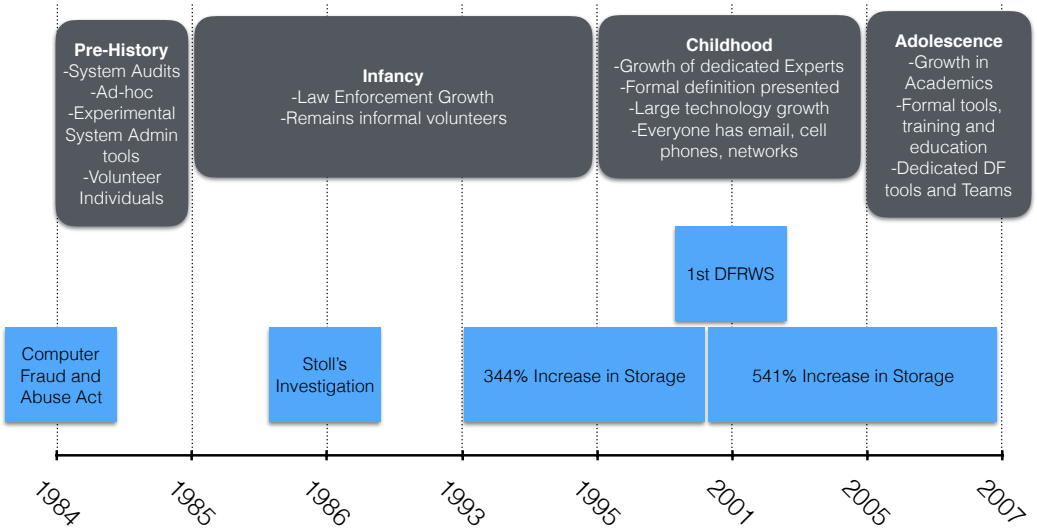

Figure 2.1. Overview of Digital Forensics Tool History. Illustrates common themes during each time period leading to the current state. As you progress from left to right, tools and digital forensics become more formalized with dedicated tools and training.

During this "Infancy" period of 1985 to 1995, as Pollitt labels it, we see a large growth of digital forensics in the Law Enforcement community a group already familiar with standards for how evidence must be collected, preserved, and presented. With agencies in this group such as the Internal Revenue Service (IRS) and Federal Bureau of Investigation (FBI) all creating computer investigation teams and conferences [5] we see how terms such as "preservation," "collection" "documentation," "presentation," "evidence" and "reconstruction" appear in the definition of digital forensics. Many of these agencies have a strong

background in traditional forensic science which is defined using many of the same terms. Though many of the tools of this period still were home-grown, the practice of applying them to digital forensics, as well as the introduction of commercial digital forensic products, became more widespread during this time [5].

In the "childhood" era, from 1995 to 2005, we see an "explosion of technology" become the primary driver for the increased need for dedicated digital forensic experts and tools [5]. Early in the era, we see greater law enforcement involvement due to an uptick in child pornography cases. But it is not until the turn of the century where we start to see explosive growth in technology. In Hilbert and Lopez's review of the world's storage and communication capacity they estimate a 344% growth between 1993 and 2000 [7]. The following period, 2000 to 2007, had a growth of 541% [7]. This growth, or turning point, in technology is significant because it shows that storage capacity and communication moved from exclusively corporate and law enforcement organizations to everyone. During this turning period in 2001, at the first Digital Forensic Research Workshop (DFRWS) [8], Digital Forensics was formally defined as: "The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations." As Hilbert and Lopez observe, the "Internet revolution began shortly after the year 2000," and this "multiplied the world's telecommunication capacity by a factor of 29" [7]. This explosion of technology was the driving factor behind digital forensics gaining importance in criminal investigations. This was because "everyone had an email address, a cell phone, [and] relied on the Internet, and most homes and businesses had networks" [5].

Starting in 2005, digital forensics moves into what Pollitt labels its "adolescence period" [5]. The previous era can be categorized as the realization and recognition of the need for digital forensics and consequent development of requirements in the field. The adolescence period is where we see those requirements come to fruition with an explosive growth into the academic community. This period marks a point where research funds are dedicated to digital forensics and universities and vendors everywhere are offering formal training. In addition, the volume of examined data reaches petabyte scale, as was the case with the FBI's Computer Analysis and Response Team (CART) [5]. Garfinkel describes this period as the

"golden age" with regard to growth in professionalization. An updated count shows there are now 16 universities offering certificate programs, five offering associates degrees, 16 offering bachelor programs, 14 offering masters, and three doctoral programs, according to the Digital Forensics Association [2], [9] .

The result of more research funds and formal programs has been the development of two primary methods to examine digital evidence: *file-based* and *bulk data analysis.* File-based tools are widely used by examiners because they are easy to understand. These tools can parse the file system, the partition table of a single pdf header and "operate by finding, extracting, identifying and processing files" [10]. These tools rely heavily on the specific metadata of the filesystem which can limit the available pieces of digital evidence they may process. A complementary method is *bulk data analysis*, which does not require knowledge or metadata from the filesystem. In contrast to its counterpart, bulk data analysis examines data of interest by scanning for content, not files [10]. This data is extracted and reported as necessary regardless of being associated with a complete file or not. The advantage to this method is that it allows tools to examine any digital storage image. *File Carving* is a specific example of bulk data analysis although it only extracts content that can be assembled into files [10]. Between these methods, we chose the bulk data analysis method and specifically the tool bulk_extractor to port to a MapReduce cluster.

2.3 Digital Forensics: Tools and Terminology

A review of digital forensics tool history is a broad topic to cover. In this section we focus on the tools and concepts that used throughout this research. The first is a review of what bulk_extractor does since this is the tool we use to extract email addresses in a MapReduce environment. Then introduce a digital forensics file format and the data set chosen for this research. Finally, a review of a term weighting concept used to analyze results is introduced.

2.3.1 bulk_extractor

Bulk_extractor is a digital forensics program written in C++ that extracts features such as email addresses, credit card numbers and URLs. A *feature* is a *pseudo-unique identifier*, such as an email Message-ID, that has been extracted from digital media. We define *pseudo-unique identifer* as "an identifier that has sufficient entropy such that within a given corpus

it is highly unlikely that the identifier will be repeated by chance" [11]. Extracted features are stored in feature file lists and may also be used to create histogram files which provide added value for a forensic analyst attempting to determine what the disk image was used for. Because bulk_extractor ignores file system structure, it can implement a highly parallel approach to processing different parts of the image. This process of dividing up a disk image is what gives bulk_extractor its performance advantage. Further, this characteristic makes it an excellent candidate for use with a MapReduce paradigm.

2.3.2 E01 and libewf

"E01" files or Expert Witness File (EWF) format is considered by many to be the de facto standard for disk images. This file format is proprietary and owned by Guidance Software who develops the digital forensics tool EnCase. This file format is used to create a compressed bit-by-bit copy of a disk prefixed with "Case Info" header and checksums for every 32KiB as well as a MD5 checksum for the entire bitstream copy [12]. Though this file format is proprietary, the open source community has reverse-engineered the software to create the libewf library [13]. The libewf library contains several tools for working with EWF files, such as `ewfacquire` to write data from drives to EWF files and `ewfexport` to export data in EWF files to raw format.

This format allows for the image to be broken up into multiple manageable segment files that can be stored across storage media that are individually smaller than the complete original drive size. The `ewfacquire` man page states that segment file size defaults to 1.4 GiB and can be controlled at acquisition time with a max size of 7.9 EiB. The first segment file is always .E01 with subsequent segment files being .E02, .E03 and so forth for the complete drive image.

2.3.3 Real Data Corpus

The Real Data Corpus (RDC) is a collection of devices including hard drive images, flash memory drives and CDROMs [14]. The devices in this collection were purchased on the secondary market in non-United States countries across the world. Images in the RDC are bit-by-bit copies of the drives when they were acquired and therefore may contain a wealth of information that forensic analysts may find on hard drives.

The uncompressed size of the RDC used in our research is 116TiB with 3,096 separate images, each ranging in size from a few GiBs to one TiB images. The RDC is the data set we chose to develop our tools against. The importance of this is that the RDC contains actual or "real" images of devices used by humans. It is important to develop our tools against a data set size and content that may actually be encountered by forensic analysts to make sure they perform correctly in real-world scenarios.

2.3.4 Term Frequency-Inverse Document Frequency

In digital forensics it is often critical to quickly determine whether a forensic artifact is significant to the case. One such method, borrowed from text mining techniques, is Term Frequency-Inverse Document Frequency (TF-IDF). TF-IDF is a statistical weighting method developed primarily for information retrieval that provides a weight value to illustrate how strongly a word is correlated to a document in a corpus [15] [16]. TF-IDF is composed of two parts; the TermFrequency(TF) part and the Inverse Document Frequency(IDF) part. TF is the number of times a word (or, in this application, an email address) occurs in a document (or forensic image). IDF is the logarithm of total documents in the corpus divided by number of documents that contain the word. The TF-IDF weighted value of an email address is the product of the TF and IDF, and describes the strength of the relationship between a particular email address and a disk image in the RDC.

2.4 Big Data

The term, "Big Data," means different things to different audiences. A generic definition of Big Data suggested by Sam Madden is "too big, too fast, or too hard for existing tools to process" [17]. This definition leaves considerable room for interpretation, however. In 2001, Doug Laney proposed the widely-cited definition for Big Data which depends on the three V's [18]:

Data Volume: The size of the dataset.
Data Velocity: The rate of flow or how fast data is produced or processed.
Data Variety: The type(s) of data in the dataset.

Laney's three V criteria remain in use today, with Gartner reiterating them in 2012 [19] [20], though NIST has proposed adding one more V: Variability (i.e., The change in velocity of

structure) [21]. The three V definition still depends on the time period: volume today may not be considered "too big" 10 years from now with tools available then. Today, large data volumes would be considered hundreds or thousands of terabytes (TB) or petabytes (PB) and in some cases even exabytes (EB).

Two prominent challenges exist with big data sets: storing the data and processing the data when, when these tasks exceed the limitations of traditional file systems and computers. An example data set, the CERN Large Hadron Collider(LHC), generated over 100PiB in 2013 with the bulk archived to tape. Though, 13PiB of it is stored on a disk pool system [22]. Storage of a data set this size requires some sort of parallel distributed file system due to traditional file system and hard drive limitations. The next challenge is processing the data, but the traditional strategy of "bring the data to the code" is simply not feasible with large data sets [23]. Big data storage and processing solutions, discussed in later sections, such as Google File System (GFS), Hadoop Distributed File System (HDFS), Hadoop and MapReduce (MR) are designed to address these storage and processing challenges with a "bring the code to the data" approach [23]. The big data approach assumes that size of processing code is drastically smaller than the data set, which is true in most cases.

Current digital forensics tools are not capable of processing large collections of disk images, such as the RDC, in an acceptable time period. Time to process larger volumes of digital evidence has remained the same or decreased during investigations. Common digital forensics tools EnCase [24], Forensic Took Kit (FTK) [25] and The SleuthKit (TSK) [26] are all considered traditional digital forensics tools capable of using multiple cores of one highly specialized machine using the "bring the data to the code" approach. Of these tools, FTK is the only that offers a distributed approach, but it is limited to a maximum of four specialized computer processing systems, one master and three workers. The three worker systems are similar to the traditional FTK install except they are installed in distributed mode [25]. These three additional systems function as workers for the primary master, which functions as the head of distributed processing.

2.5 MapReduce Paradigm

Analyzing a large data set using traditional computing methods is not feasible. Even with increased multiprocessing capabilities common digital forensics tools are still not sufficient

for the data sets of today. Today's tools are still predominately limited to one highly specialized computer and even with continued advances in CPU speeds they are still limited by I/O speed, which has seen far less impressive performance improvements [1]. Roussev *et. al* argue that successful next-generation digital forensics tools will employ methods to distribute the I/O limitations to multiple machines capable of potentially processing tens of thousands of image thumbnails [1], [27].

One possible distribution method is MapReduce. MapReduce is a scalable tool capable of processing large data sets using low-end computer systems in parallel [28]. The MapReduce programming model was originally developed at Google and later became the basis the open source version, Hadoop [29], [30]. This programming model follows a divide and conquer approach, which breaks up a large job (i.e., dataset) into smaller chunks that are then processed in parallel.

A MapReduce job is composed of two functions: Map and Reduce. Appendix D contains pseudo-code for these functions taken from Dean and Ghemawat's work [29]. The Map function is written by the user and takes an input pair and produces a set of intermediate key/value pairs. Prior to passing all intermediate pairs to a reduce function the key/value pairs are grouped on the same intermediate key, then all pairs with same intermediate key are passed to the same reduce function. The Reduce function, also written by the user, takes as input the intermediate key and its values, which are merged to produce a smaller set of values. Map functions are executed in parallel across a cluster of machines by partitioning the input data into a number of input splits. The number is typically driven by the block size of the data stored in Hadoop Distributed File System (HDFS). The number of Reduce functions is controlled by the user specifications, which are also executed in parallel across a cluster.

An overview of MapReduce job execution is shown in Figure 2.2 from Dean and Ghemawat [29]. We summarize their description in the following steps:

1. MapReduce splits up the input files into chunks based on their HDFS data block size. The default data block size is 128 mebibytes (MiB) [31], but can be controlled by the user upon importing data files into HDFS.
2. Once the number of input splits (also equal to the minimum number of mapper tasks) and reducer tasks is determined, the master assigns these tasks to idle datanodes in

the cluster. The master is aware of which datanodes contain which chunk of the input files and will make the best effort to assign mapper tasks to those datanodes. This strategy achieves the goal of data-local execution, which we expand on later.

3. A worker is assigned a mapper, which reads the contents of the input split, parses the key/value pairs, then executes the user-defined map function on the value.
4. The intermediate key/value pair result from a mapper is written to the local disk and the worker notifies the master of this location. The master then sends the location to workers assigned a reducer task.
5. A reducer task performs a remote read of this intermediate file where it is sorted by intermediate keys. It then executes the user-defined reduce function on this sorted data.
6. The reduce function output is written to the final output file.

The MapReduce program is complete once all map and reduce tasks are finished with reduce output files containing combined results.

We chose MapReduce processing and the Apache Hadoop Framework for our research because it provides a parallel processing solution for data-intensive applications. Furthermore, a MapReduce program is naturally parallel [32] thus eliminating the need for the user to struggle with the details of parallelizing the process [28]. Thus MapReduce provides parallel data analysis access to any forensic analyst with little to no parallel programming knowledge. Additionally, MapReduce and the Apache Hadoop Framework provide a cost effective solution to parallel processing compared to traditional High Performance Computing (HPC). The Apache Hadoop Framework is a cost effective solution for a few reasons: it is open source with no licensing costs, installed on inexpensive commodity hardware and requires minimal training to write Map and Reduce functions.

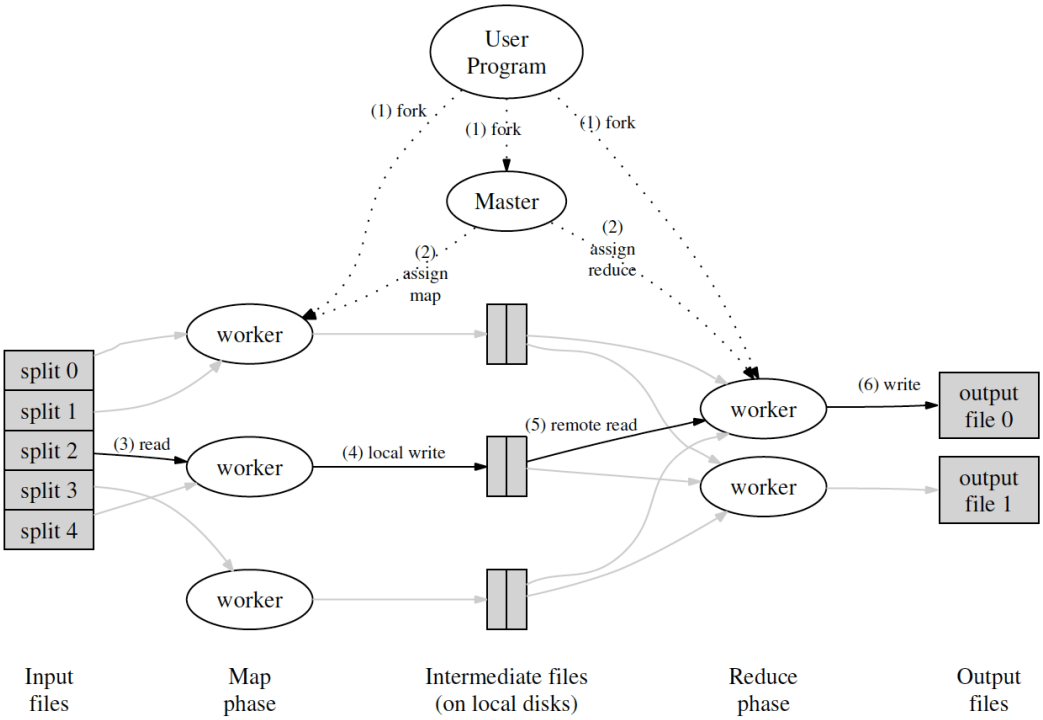

Figure 2.2. Overview of the Execution of a MapReduce Program. Starting from top to bottom then left to right with InputSplits of a file the figure proceeds through the execution of the map phase, where the output is written to local files. These files are read remotely in step 5 by the reducer, which executes the reduce phase on them producing final output for the program. Source: [29].

2.6 Hadoop and HDFS Architecture

Before creating the MapReduce programming model, a team at Google designed a distributed file system called Google File System (GFS). GFS is a reliable file system distributed across commodity hardware used for large data set analysis [33]. The Hadoop framework is the open source software implementation of both the GFS and MapReduce concepts. We have already discussed the details of MapReduce processing and here we will discuss the architecture of HDFS and the specific implementation at the Naval Postgraduate School (NPS).

Next-gen digital forensics tools need to deploy distributed methods for both storage and processing. MapReduce handles the processing aspect while HDFS handles the storage aspect. The significance of a distributed parallel file system, such as HDFS, to our research is that it distributes the I/O limitations of current digital forensics tools to several other computer systems. HDFS's purpose is to allow multiple chunks of a single input file to be read and processed, using MapReduce, in parallel, therefore distributing I/O across the cluster and decreasing overall processing time.

Similar to GFS, the design goals of HDFS are: store very large files, implement a write-once, read-many-times pattern, and use commonly available hardware with high node failure rate [32], [33]. Like many other distributed file systems, HDFS stores metadata and application data separately. HDFS uses a NameNode and DataNodes to store metadata and application data, respectively. This can be thought of as a Master and worker relationship as shown in Figure 2.2.

A Hadoop cluster consists of a minimum of one NameNode and potentially tens to thousands of datanodes, though a secondary NameNode may also be used as backup due to its crucial role in the architecture. Unlike other distributed file systems, HDFS uses block replication for data protection against node failure. This different approach provides durability and more opportunities for computation near the data, which is critical for data local computing [34]. Specifically, HDFS by default uses large 128MB block sizes and replicates each block of data three times across datanodes in the cluster, though these values can be set by the user on a file-by-file basis [34]. Figure 2.3 is an illustration of this replication and block size for a sample file. A Real Data Corpus (RDC) [14] 160 GiB disk image stored with a 512MiB block size would be replicated across three datanodes. We discuss in Chapter 4 why the 512MiB block size was chosen.

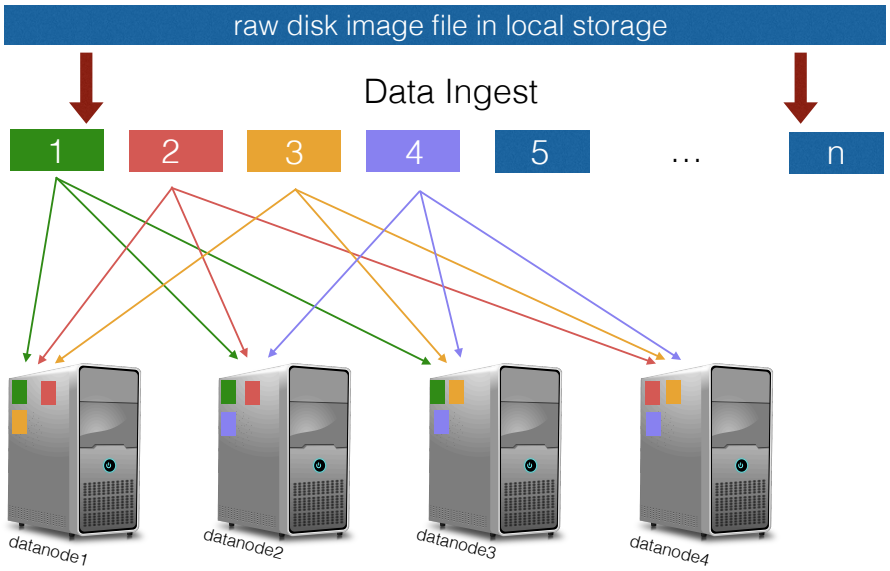

Figure 2.3. File Representation in HDFS. During data ingestion into HDFS a file is broken into chunks. These chunks are then replicated by default three times across the available datanodes in HDFS. By default, three copies are made. For example, the green chunk in the figure above is replicated across datanodes 2 and 3 to preserve fault tolerance in the case datanode1 fails.

The Hadoop cluster NameNode manages HDFS attributes such as location of chunks and their mapping to files as well as permissions; therefore any read request for a file must first go through the NameNode [33]. It is important to note that the NameNode manages HDFS all in memory, thus the NameNode is likely to have more memory and smaller hard drive space than datanodes. This design inherently creates some degree of overhead during set up of a MapReduce program and is one reason why we chose a 512MB block size, since this choice reduces the number of map tasks the NameNode has to manage. The NameNode receives updates from all datanodes in the cluster via heartbeats and if NameNode fails to receive an update in ten minutes it marks this datanode and the block replicas as unavailable. Because there are two other replicas, any jobs on this datanode will be resubmitted to other datanodes where a replica exists while the NameNode will schedule the creation of new

replicas to maintain three replicas.

2.7 HPC and MapReduce Trade-offs

Distributed processing and storage typically means one of two approaches: High Performance Computing (HPC) or MapReduce. Both attempt to solve complex large problems but with subtle differences. HPC aims to solve problems that require extensive computation of complex equations on a dataset that is potentially, but not typically large. MapReduce aims to solve problems that are data-intensive. Generically, HPC solves problems that are seen to be CPU-bound while MapReduce solve problems that are seen to be I/O-Bound. In their attempt to solve different complex problems each make intentional trade-offs, which we review in this section.

Traditional High Performance Computing (HPC) uses many high end nodes in parallel to run advanced applications that are not capable of running on a single system. HPC originated from the need for compute-intensive applications [35]. It relies heavily on a centralized parallel file system that is accessed by compute nodes with limited local storage via a high-end network using technologies such as Infiniband. Reliance on this parallel filesystem and accompanying high-end network can have high costs up front as well as high maintenance especially when looking to scale. That is not to say HPC is obsolete; in fact, in recent years HPC has made significant advancements in terms of compute-intensive applications, just as Hadoop has done with data-intensive applications.

A serious comparison of HPC and MapReduce must distinguish between the fundamentally different approaches taken by each in terms of data locality. It is trivial to see with HPC's central parallel file system that locality does not exist where computations occur. That is storage of data is separate from computing on the data and HPC uses a "bring the data to the code" method [23]. With smaller data sets and high-end networks, this approach works, but with larger and growing datasets (e.g. Real Data Corpus, 116TiB) this move takes time even with high-end networks. MapReduce approaches the problem with a "bring the code to the data" method. The code of data-intensive applications is much smaller than the data they are processing.

Another important difference between the two approaches is resource scheduling and fault tolerance. HDFS achieves fault tolerance by using block replication across datanodes,

whereas HPC typically uses redundant array of independent disks (RAID). Both implementations have drawbacks, RAID large capacity rebuilds can days to weeks whereas block replication requires two or more times storage space [36]. The additional storage capacity requirement for replication is mitigated by low cost of hard drives. Because of these different strategies, the impact of a node failure on a given active job is also different. With HPC, many times node failure means the whole job must be started from the beginning, whereas with MapReduce and its YARN (Yet Another Resource Negotiator) task scheduler only those mappers or reducers whose data block is on the failed node must be submitted using one of the replicated blocks [37].

From a user's perspective a significant difference between HPC and Hadoop MapReduce is the level of programming knowledge required. HPC programming uses Message Passing Interface (MPI) libraries, which puts the responsibility on the user to "manage communications, synchronization, I/O, debugging, and possibly checkpointing/restart operations" [37]. A MapReduce programmer is only provides a Map and Reduce function. This represents a significant reduction in the complexity of the interface.

CHAPTER 3:
Related Work

Our research seeks to provide a scalable Digital Forensic tool using the MapReduce framework. To do that we build upon previous research and digital forensic tools, as well as work that attempts to improve performance with distributed solutions. In this chapter we review the previous work from these perspectives:

1. Does it provide a performance solution to increasing data set size?
2. Does it add more complexity to the tool for analysts?
3. Is it tested against a large dataset(>1TiB)?
4. Are the costs acceptable? In this context, cost is not limited to monetary cost but includes cost of knowledge required for an analyst.

See Table 3.1 for an overview of these answers. We divide our review of previous work into three categories:

1. Current state of digital forensics tools.
2. Requirement for distributed tools.
3. Existing distributed approaches.

Table 3.1. Summary of Related Work

Solution	**Does it improve performance for large dataset?**	**Does it add complexity?**	**Is it tested against large dataset (>1TiB)?**	**What are the costs?**
Roussev Breaking Performance Wall (DELV)	Yes	Yes	No	Specialized knowledge and monetary cost of high-end network switch
EnCase	No	Yes	No	Highly specialized knowledge and restricted to specific media and file formats
Sleuthkit	No	Yes	No	Highly specialized knowledge and restricted to specific media and file formats
Forensic Toolkit (FTK)	Yes, with ProcessManager	Yes	No	Highly specialized knowledge and restricted to specific media and file formats, limited to maximum of 3 specialized machines
pyflag	No	Yes	No	Highly specialized knowledge and restricted to specific media and file formats
Data Reduction	No	No	No	Lost evidence
MPI MapReduce (MMR)	Yes	Yes	No	MPI knowledge and complexity required for analyst
Sleuthkit Hadoop	Inferred, no evidence to support	Yes	No	Not enough info on solution. Incomplete implementation.
Massive Threading w/GPUs	Yes	Yes	No	Complexity and knowledge cost to learn GPU programming, monetary cost of specialized GPUs
Parallel GPU in GPU memory	Yes	Yes	No	Complexity and knowledge cost to learn GPU programming, monetary cost of specialized GPUs
Tarari Processor	Yes	Yes	No	Complexity and knowledge cost to learn Tarari processor programming
HPC password cracking	Yes	Yes	Not applicable for password cracking	MPI knowledge and complexity required for analyst
z-algorithm search w/MPI	Yes	Yes	Not Applicable	MPI knowledge and complexity required for analyst
forensic cloud	Yes	Yes	No	MPI knowledge and complexity required for analyst
Netherland XIRAF/HANSKEN	Yes	Yes	Yes	Complexity and knowledge of custom XIRAF system
DFaaS Index Search MapReduce	Yes	No	Yes	Added complexity for web-based interface, but not backend MapReduce Computations

This chapter is not an all-encompassing review of digital forensics tools. Rather, we provide a progression of digital forensics tools research that aims to solve the volume problem and improve performance. We start with the current state of digital forensics tools, most of which are single system bound. We then review research that examines and defines a requirement for distributed solutions. The closing sections are more directly related to our research and review specific attempts to distribute digital forensics tools and several data mining approaches used.

3.1 Current State of Digital Forensics

We begin with a review of the current state of digital forensic tools, their successes as well as their limitations when attempting to analyze a data set size of 116TiB, such as the Real Data Corpus. The volume increase in Digital Forensics has had dramatic impacts on the research and tools used. Quick and Choo provide a thorough survey of these impacts as well as proposed solutions but group distributed processing with other topics that appear to require additional research [38].

One of the earliest works to recognize the volume growth challenge is Richard and Roussev's paper in 2004, after observing that FTK takes 60 hours to open an 80GB case [1]. They do not attribute this performance to bad implementation of the FTK tool. Rather, they view their results as a warning sign to all digital forensics tools. Richard and Roussev state that a specialized solution is better than a generic distributed one and developed a prototype, DELV, with an approximate 2600% time improvement over FTK. In 2006, Richard and Roussev describe what the next generation of digital forensics tools must accomplish, primarily focusing on the issue of scale due to data volume and the fact that current tools are constrained to a single workstation, which are bounded in computation cycles [27]. Their prototype, DELV, demonstrates the benefits distributed digital forensics has over current tools. The performance improvements of this prototype come at the cost of increased complexity, monetary expense and knowledge it requires for any analyst to implement and use. Specifically, considerable expertise is required to understand and learn their custom communication protocol. The authors mention that no specialized libraries such as PVM (Parallel Virtual Machine) or MPI (Message Passing Interface) were used, but the creation of a specialized solution adds to complexity and to the knowledge the analyst must acquire. Richard and Roussev's work provides the foundational case against current digital forensics

tools and the DELV prototype provides a coordinator/worker architecture we build on.

The DELV prototype provides a great starting point, and in 2009 Nance *et al.* define a research agenda to address data volume [39]. This trend continues for the next several years. In 2010, Garfinkel states in his next 10 years of research that some tools today can process terabyte sized cases, but are unable to create a concise report of features [2]. We add that those tools capable of working with a terabyte case can take days to weeks on a single specialized workstation. His point is further illustrated with Raghavan's 2013 review of current digital forensics research stating that popular digital forensics tools, EnCase, Sleuthkit, FTK and pyflag are "highly specialized" and "fine-tuned" to specific storage media and fail to address the "single largest challenge" going forward, data volume [40]. In 2015, mentions of vast volumes of data as a challenge still exist with most solutions focusing on reduction of data, which could lead to missed data in an investigation [41].

The most popular tools today are EnCase [24], FTK [25], Autopsy and Sleuthkit [26]. All perform well at specific, isolated tasks, but the available detailed comparisons of the tools primarily focus on formats and file systems they can read, whether they provide searches and what searches they can perform. We recognize these are valid points to compare against, but in terms of how each perform with large data sets, little analysis has been performed. Some articles provide details of each of these tools' features [42] and some perform analysis by executing tools against actual images [43], but the images they test on are limited in size with the largest being 15GiB. Of these tools, AccessData's FTK is the only to distribute the workload with its Processing Manager, but it is limited to three additional systems [44], [45]. Though it is limited, the performance improvements show promise for distributed tools going forward.

This capability for FTK was released with version 3.0.4 in 2009 along with actual test results shown in Table 3.2. These results provide strong evidence for the case for distributed forensic solutions, but also highlight the fact that they are dated considering many modern images are much larger than 160GiB.

Table 3.2. AccessData FTK Distributed Test Results. Source: [46]

Image	Size of Data Set	Time w/Single Node	Time w/Distributed Processing
Image #1	100GB	9.08 hours	2.13 hours
Image #2	160GB	8.57 hours	1.68 hours
Image #3	140GB	13.48 hours	5.63 hours
Image #4	75GB	6.96 hours	2.75 hours

3.2 Call for Scalable Digital Forensics

Taking 60 hours to process an 80GB image is unacceptable by today's standards considering typical consumer hard drives are now measured in the hundreds to thousands of GBs. Digital forensic tools must advance and scale to analyze this growing data volume and continue to provide critical insights into investigations as it has in the past.

Richard and Roussev's call for distributed digital forensics in 2004 was reiterated in 2016 when Lillis *et al.* [47] categorized distributed processing, HPC and Parallel Processing in their future research chapters. As the driving justification for distributed processing, Lillis *et al.* point to Roussev *et al.* [48] 2013 findings that with current software it would require 120-200 cores to keep up with commodity HDDs. It is implied that this means multiple systems must be used since no single CPU contains this many cores. Roussev *et al.* findings build on Roussev's previous work with scalable open source tools in 2011 where he again points to a lack of distributed tools, such as Google's MapReduce Framework, to address the scalability problem, which Lillis *et al.* conclude has not received sufficient attention by researchers in digital forensics [49].

Both Richard and Roussev and Lillis *et al.* define a model to follow and tools that could potentially be used, but they do not offer implementations. Rather provides foundations to many of the implementations for scalable digital forensics tools, which we discuss in the next section. One of the models we aim to implement and build on is described in an AccessData whitepaper [50], which suggests using legacy hardware to distribute processing and reduce time. In this respect, it advocates for principles similar to those followed by Hadoop MapReduce which uses commodity hardware for parallel processing.

Similarly, Ayers' work describes a set of requirements for second generation tools which

include parallel processing and fault tolerant data storage [51]. Both are achieved with the Hadoop MapReduce implementation we propose and align with Ayers' proposal to use super computers and parallel file systems. Roussev's proposed distributed prototype, DELV, is another building block for many of the existing distributed approaches. Each of these approaches attempts to achieve many or all of the requirements Roussev defines for Distributed Digital Forensics (DDF). Those requirements stipulate that a distributed digital forensic system must be [1]:

1. *Scalable* Able to employ tens to hundreds of machines which should lead to near linear performance improvement.
2. *Platform-independent.* Able to employ any unused machine on local network.
3. *Lightweight.* In terms of *efficiency* and *easy administration* such that extra work to install, run and distribute data should be negligible.
4. *Interactive.* Capable of allowing interaction with partial results while distributed processing is executing.
5. *Extensible.* It should be easy to add new functions with little to no additional effort and skills over sequential case.
6. *Robust.* It must ensure the same level of confidence as sequential case when a worker node fails.

Our distributed approach satisfies and improves on the work Roussev laid out with DELV. The achievements of our work are based on these requirements, which we review and discuss in later chapters.

3.3 Current Attempts to Scale and Distribute

This section reviews existing distributed approaches to digital forensics. We divide these existing approaches into four categories and review what each attempt has accomplished:

1. MapReduce Attempts
2. Hardware Attempts
3. HPC Attempts
4. Cloud-Based Attempts

Each of these attempts is examined based on the previously defined questions:

1. Does it provide a performance solution to increasing data set size?
2. Does it add more complexity to the tool for analysts?
3. Are the costs acceptable?

3.3.1 MapReduce Attempts

MPI MapReduce (MMR) [52] relates directly to our research. Roussev *et al.* develop a platform to use MPI with MapReduce that achieves super-linear speedup for indexing related tasks, linear speedup for CPU bound processing and sub-linear speedup for I/O-bound tasks. These results show great promise and inspire confidence in using MapReduce for digital forensics related tasks. Though this work produces promising results, it is implemented on a three node cluster (12 cores total) and only tested against relatively small files (less than 2GB). Furthermore, it adds a layer of complexity: the analyst is expected to have a "fairly good understanding of distributed programming" [52]. The addition of MPI complexity does not meet the extensibility requirement, and implementation on three nodes does not demonstrate scalability.

In addition to the above MPI MapReduce implementation, another project that uses the MapReduce paradigm to improve performance is Sleuth Kit Hadoop. The goal of this project is to incorporate The Sleuth Kit into a Hadoop cluster [53]. Initial efforts were funded by U.S. Army Intelligence Center of Excellence, but there has not been an official release, and the latest source code commit coming in 2012. Miller *et al.* [54] state this project has three phases — ingest, analysis, and reporting— but do not mention any empirical results, only that TSK and Hadoop "together benefit from increased processing power from parallelization." The authors did not attempt to install this framework, but felt it significant to mention as related work using MapReduce in a digital forensics setting. Even without official release and frequent updates, this project shows potential to provide a lightweight solution that satisfies the scalability, platform-independence, lightweight and robustness requirements.

Hadoop and MapReduce Scalability

MapReduce has the ability to easily add datanodes, extending HDFS capacity and processing on data stored in HDFS. Gunther *et al.* state that the Hadoop framework ensures that MapReduce applications shown to work on a small cluster(less than 100 datanodes) can

scale to arbitrarily large clusters (several thousand datanodes) [55]. Many corporations maintain Hadoop clusters containing several thousand data nodes with Petabytes of HDFS capacity, but the question of scalability for distributed forensics depends on whether it provides near linear performance improvement.

Various related works on Hadoop scalability found great performance benefits are achieved by scaling Hadoop clusters, though many suggest these performance improvements depend on the application. For instance, Appuswamy *et al.* [56] test 11 different jobs and find that doubling cluster size improves performance for six jobs. Generally, they conclude that scaling out, or adding more cluster nodes, worked better for CPU-intensive applications.

Furthermore, Li *et al.* [57] compared scale-out and scale-up strategies for HDFS and local file systems. Scaling-up means improving the components in existing datanodes instead of adding more datanodes. Li *et al.* conclude that scaling out performs best for I/O-intensive applications with small files and that scaling out HDFS outperforms scaling up HDFS in three of the seven applications tested. Overall, Hadoop scaling provides application speedup, but predicting the exact performance improvements for a given job remains difficult.

3.3.2 Hardware Attempts

Massive multi-threading [58] attempts to utilize new hardware graphics processing units (GPU) to improve file carving. Marziale *et al.* results demonstrate significant performance improvements executing Scapel file carver with GPUs versus multicore CPUs. Scapel [59] is a file carver that extracts files based on known byte patterns. This is done without assistance from the file system, which gives it the ability to extract data from unallocated spaces in the file system. Specifically, a 150% speedup is measured using a massively multi-threaded GPU-enabled scalpel over a multicore-threading CPU-enabled scalpel running on a 100GiB disk image. These results show promise for using GPU-based solutions in digital forensics, but the authors point out that there is added difficulty in GPU programming [58]. Therefore, while this approach does improve performance, it does so at the cost of added complexity and specialized knowledge. We therefore argue it does not meet the extensibility requirement. Additionally, this work is limited to a single workstation with one GPU, which requires additional storage and may degrade performance when processing a dataset such as the RDC, where disk images may range up to 1TiB.

Additional work with GPU's is explored in [60] and performance gains from parallelizing hash-based data carving. The work of Collange *et al.* in parallelizing hash-based carving demonstrates that the most effective use of GPU's occurs when data is stored in the GPU memory as opposed to main memory or on disk. Their results illustrate the benefit of parallel GPU processing over serial CPU processing and build upon the results from Marziale *et al.* This comes at the monetary cost of GPUs as well as knowledge and complexity costs associated with GPU programming.

Lee *et al.* [61] propose another hardware based approach to improve search performance in digital forensics. They propose using the Tarari content processor for improved search. The implementation improves performance by a factor of five over tools such as EnCase. Though this implementation, similar to GPU approaches, improves performance it remains limited to a single workstation and was tested only on small file sizes. Similar to GPU programming, we argue the performance gains are negated by the complexity required to program new functions and to actually scale to a full dataset the size of the RDC.

3.3.3 HPC Attempts

High-performance computing (HPC) clusters are alternatives to MapReduce's distributed parallel processing. HPC uses a model to divide a data set into smaller parts and share the workload amongst multiple cluster nodes communicating via some message passing technique. Bengtsson [62] provides a survey of how an HPC Linux cluster is used to speedup password cracking. This work demonstrates a type of forensic problem that can be divided up and leverage HPC for performance gains. Early work with HPC demonstrates its possibilities to improve current digital forensics tools performance, but require a great deal of specialized of knowledge to program. In addition, scaling to large datasets is expensive.

Additional HPC work done in uses MPI calls to improve performance by a factor of six for *z* algorithm, a linear time pattern searching algorithm [63]. This work provides solutions for the growing data volume problem but require an analyst with extensive MPI knowledge.

3.3.4 Cloud-Based Attempts

Another avenue being explored to combat the data volume challenge is Digital Forensics as a Service(DFaaS) using a cloud platform. Miller *et al.* [54] reiterate NIST's cloud definition

to be, "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources." Miller *et al.* define the process, model and flow of a "forensicloud" where they test performance of a workstation cluster, virtualized cluster and HPC backend for processing compared to a single node of each type. Results from this illustrate distributing processing workload reduces run time by 50% for each cluster type. Miller *et al.* conclude that the virtualized cluster performed the best, but attribute this to HPC lacking fast access storage device similar to the virtualized cluster. We point out this research was tested with a single file of 40GiB uncompressed which is an improvement over previous tests but still needs further testing considering today's standard is 500GiB or 1TiB hard drives.

Similar cloud forensics work is being performed at the Netherlands Forensic Institute (NFI) where they are using Hadoop MapReduce for backend processing. Much of this work is based around an XML Information Retrieval Approach to Digital Forensics (XIRAF) [64], [65], [66] system and its successor HANSKEN. The initial design for XIRAF was not to process petabytes of data and as such required improvements when it attempting to [67]. Predominantly improvements in the extraction service to now use Hadoop MapReduce to drastically reduce time from 24 hours per terabyte to three terabytes per hour. In van Beek's 2016, update a key lesson from implementing MapReduce was to "bring computing power to the data" [68]. This work is similar to achievements our work provides, but is focused on providing a front end cloud service.

A case study of DFaaS [69] found linear speedup proportional to the number of datanodes in a Hadoop cluster when processing large datasets, larger than 56GiB. Lee *et al.* report the ability to perform bigram frequency analysis on 1TiB of data in about 2 hours. Their work demonstrates advantages MapReduce has when applied to a web-based search service, which allows for remote upload of an analyst's image file. The backend MapReduce analysis portion of this work aligns directly with our research to use MapReduce to improve performance digital forensics tools.

Another project that describes itself as "running forensic workloads on cloud platforms," is a platform developed at Google called Turbinia: Cloud-scale forensics. There is not much published on this project, only the source code, which indicates reliance on an existing Hadoop cluster [70].

Several of the works mentioned above attempt to solve the growing data set problem, but our literature review found many fall sort of providing empirical evidence their solutions succeed with large data sets. Several use various MapReduce implementations to meet distributed digital forensics requirements. Additionally, some solutions come with a cost of increased analyst knowledge of specialized fields such MPI programming and hardware component programming.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Methodology

Many of the previous attempts to create scalable forensic solutions mentioned in Chapter 3 are developed using small sample files to perform experiments. Though prior work to improve and ultimately distribute digital forensics tools has shown promising results, many add complexity for the analyst or have not been tested against large datasets such as the Real Data Corpus (RDC) (see 2.3.3.) This work builds on these prior results. Additionally we chose to not use complex programming languages such as hardware chip programming or MPI that require additional skills and training for the developer.

To demonstrate the advantages of our approach, we develop a tool for scalable and execution of a bulk_extractor email scanner on a Hadoop cluster and measure tool performance for processing large disk image data sets. Specifically, our tool adapts the Hadoop platform to large-scale forensic analysis. The MapReduce paradigm requires the analyst to only develop two functions, a Mapper and a Reducer, using a common language, Java. These functions, in combination with our InputFormat Class for binary images, are sufficient to demonstrate successful execution of a massively parallel bulk_extractor on the RDC with significant performance improvements.

This chapter describes the steps taken to develop and implement our solution. First, proof of concept work is done via a virtualized Hadoop cluster. We then perform a review of the requirement to develop a custom InputFormat class to read and process binary images in HDFS. Next, we convert the RDC from E01 format to raw format and import it into HDFS. The reason for this conversion is that raw is a much simpler format to work with when developing an InputFormat class to read this data. After this, we develop a byte counting program that we use to tune MapReduce parameters as well as the InputFormat Class. Finally, we successfully write and run a MapReduce bulk_extractor email scanner against RDC data set on NPS Hadoop Cluster, Grace.

4.1 Infrastructure

This section discusses details of hardware and software used to perform our experiments.

4.1.1 Virtualized Hadoop Cluster

During the research design phase we built a small virtualized Apache Hadoop cluster for preliminary testing. This cluster was built in NPS' DEEP laboratory and consisted of one NameNode and six slave DataNodes each with 1GiB of RAM and minimal hard drive space. Each of these nodes had Centos 7 installed and Apace Hadoop 2.6.0 installed following Apache's install steps [71]. This virtualized cluster was built to demonstrate proof-of-concept of early designs, therefore performance gains or losses were not measured.

One of the early questions when examining MapReduce for distributed digital forensics was how binary disk images should be stored in HDFS. The format of the files in our dataset are raw bitwise images of hard drives, which is common practice in the digital forensics community, but not the default or ideal format for MapReduce jobs. This motivated writing the custom InputFormat class, discussed further in Section 4.2.2.

4.1.2 Grace Hadoop Cluster

The Hadoop cluster at the Naval Postgraduate school, "Grace," contains 2 NameNodes with 24 datanodes for processing. Each NameNode contains 252 GiB RAM and 40 processors with hyper-threading. Datanodes contain 504 GiB RAM, 24 processors with hyper-threading, and 12 5.5TiB hard drives for a total HDFS capacity of 1.5PiB. The cluster configuration is illustrated in Figure 4.1. However, each datanode is configured to make only 256GiB of RAM and 80 vcores available for MapReduce jobs. This means there are 1,920 (80×24) task slots in our cluster meaning at any one time there can be only 1,920 Mappers or Reducers executing. Therefore, if a MapReduce job has more than 1,920 tasks, Mapper or Reducers, they will complete in waves.

4.2 Preparation

This section discusses preparatory steps once a Hadoop Cluster is installed before experiments can be performed. We provide details on the dataset format and development of necessary code, including rawInputFormat and two MapReduce programs: byteCount and bulk_extractor MapReduce.

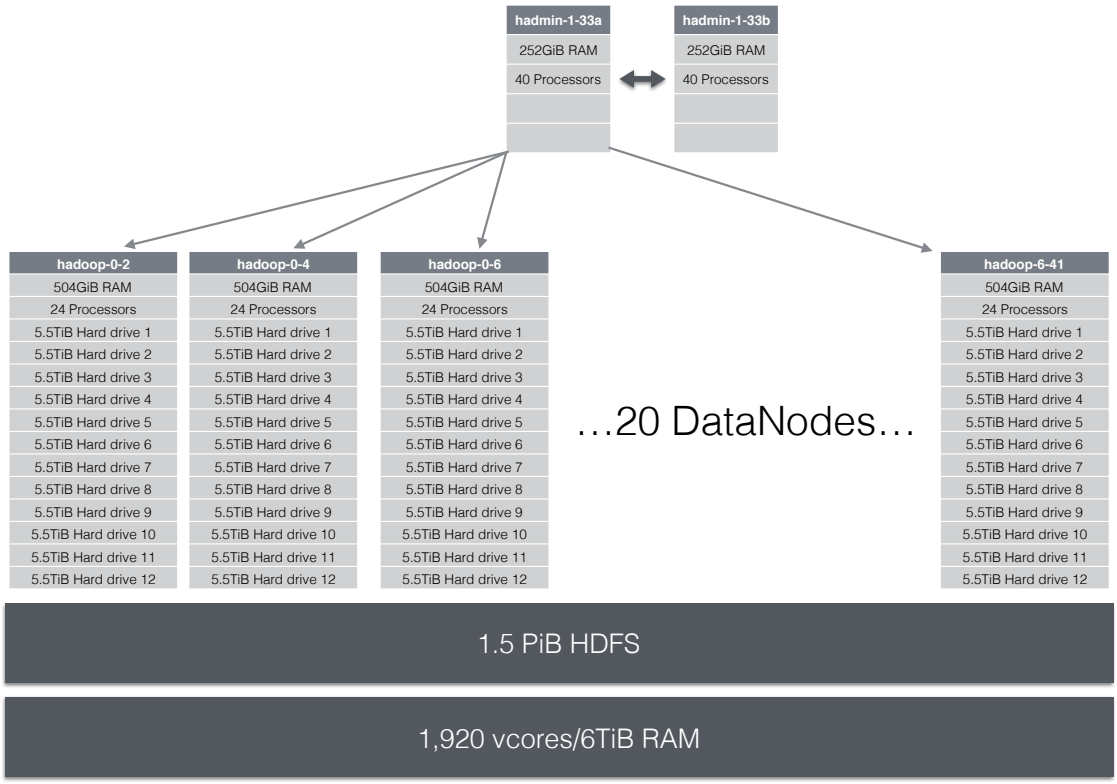

Figure 4.1. NPS Grace Cluster Architecture. Grace cluster at NPS is configured with 24 datanodes, each with 504GiB RAM and 12 5.5TiB hard drives. However, only 256GiB of RAM and 80 vcores per datanode are available for Hadoop processing.

4.2.1 Converting E01 to Raw Images in HDFS

E01 file format is the most popular file format used for digital forensics because of its compression and segment file usage. These features are useful for storing forensic evidence, but for our purposes the format and segment files added additional complexity and processing. Therefore, it was necessary to convert the RDC from E01 files to raw format when importing them into HDFS. This was done at the expense of storage space. Raw images lose the E01 block level compression and this results in approximately 2.1 times more storage space causing an expansion from 59TiB to 116TiB. The final storage footprint to store the raw images of the RDC is roughly 385TiB when the two replicas are factored in for each block. NPS' Hadoop Cluster has 1.5PiB of HDFS storage capacity; therefore, this conversion was

not a problem.

Conversion from E01 to raw format is a time consuming process, especially for a large data set, such as the RDC. At NPS, in addition to the Hadoop cluster, Grace, there is a traditional HPC cluster, Hamming. To speed up the conversion we used Hamming to export to raw using `ewfexport` and input the raw file into HDFS on Grace in parallel. Success of this approach was dependent on both Hamming and Grace having a shared parallel file system as well as network access between them. Full HPC job scripts to do this may be found in Appendix A.

To convert the E01 files we submit a SLURM [72] job for each file in the RDC. SLURM is a resource manager commonly used in HPC clusters. It manages job submission to any of the various compute nodes in the cluster based on job resource requirements. This SLURM job creates an array of all the files in the RDC, excluding ones that were found to be corrupt and not able to be converted. This array is then used to submit `ewfexport` commands for each file with a maximum of five running at one time to avoid overloading this file system. The converted raw file is placed on a shared file system between Grace and Hamming; where an `ssh` session to a Grace node is initiated and performs an `hdfs put` operation on the converted file to import the file into HDFS. Cleanup is then performed to remove the raw converted file from the shared file system.

4.2.2 The rawInputFormat

The virtual cluster was used to initially determine that a custom InputFormat class, which we name rawInputFormat, was needed for the RDC. A detailed discussion of why and how this class was written follows below.

Why Create a New InputFormat Class?

Hadoop was originally developed for processing large quantities of text, therefore the default InputFormat is TextInputFormat [32]. A MapReduce job requires data input to be in the form of key-value pair records. How these pairs are determined is defined in the InputFormat class that the MapReduce job uses. Specifically, the RecordReader makes this determination.

TextInputFormat treats each line in the file as a record, where the key is byte offset in the file and the value is the contents of that line. Records are created via LineRecordReader,

which creates a record any time a newline or carriage return is encountered in the file. This logic is excellent for text files, but a disk image is a stream of bits that may have few or no newline or carriage return characters. For instance, a newline could be in the first several bytes of a disk image, but then not appear for the rest of the disk image. Attempts to use the default TextInputFormat and the LineRecordReader in the virtualized cluster fail when the space between newlines exceeds max record size. This max size is set to two GiB as seen in the lines of code below, taken from the Apache Hadoop source code [73]. Note that `Integer.MAX_VALUE` is 2,147,483,647 bytes.

```
1          .
2          .
3          .
4    public void initialize(InputSplit genericSplit,
5                           TaskAttemptContext context) throws
                               IOException {
6      FileSplit split = (FileSplit) genericSplit;
7      Configuration job = context.getConfiguration();
8      this.maxLineLength = job.getInt(MAX_LINE_LENGTH, Integer.
           MAX_VALUE);
9      start = split.getStart();
10     end = start + split.getLength();
11         final Path file = split.getPath();
12         .
13         .
14         .
```

This behavior is unpredictable for binary disk images without detailed inspection of the file and therefore not suitable for our use case. Several other InputFormat exists within Hadoop. However, the majority focus on processing textual data and are similarly unsuitable.

Hadoop also contains support for binary input formats. Two InputFormats support binary input: `SequenceFileInputFormat` and `FixedLenghInputFormat`. SequenceFileInputFormat and specifically `SequenceFileAsBinaryInputFormat` were designed to address scenarios where plain text is not suitable. Sequence files are flat files that consist of binary key-value pairs with some header information where keys and values are user-defined at sequence file creation time [32]. Use of this InputFormat requires some preprocessing to

convert RDC files into a sequence file format. Eventually, such a format might serve as a replacement for E01 on HDFS. A benefit of this approach is support for compression. However, we leave this for future work.

`FixedLengthInputFormat` is used to read fixed-width binary records from a file [32]. This InputFormat does not require the pre-processing that Sequence Files require. The data set may be stored as a raw binary disk image in HDFS where the only pre-processing time is the time to import to HDFS. This InputFormat requires the programmer or analyst to set the recordLength during job set up. A requirement of this InputFormat is that total file size be evenly divisible by the recordLength. Otherwise the program will throw an error. This requirement is explicitly defined in the `FixedLengthRecordReader` source code. This InputFormat comes closest to meeting our needs for binary input data without additional pre-processing, with the only drawback being that many files will not be evenly divisible by the recordLength. Review of this InputFormat led to the creation of `rawInputFormat` and `rawInputRecordReader` (See Appendix B for code) which are based on `FixedLengthInputFormat` [74].

How was RawInputFormat Developed?

Every InputFormat contains two parts: the inputFormat and the RecordReader. The inputFormat performs the following tasks for each job [75]:

1. Validate input-specification
2. Split the input file(s) into logical InputSplits, each which is assigned to a Mapper
3. Provide the RecordReader implementation to be used to create input records from the logical InputSplit for Mapper processing.

The RecordReader creates records, which are the key-value pairs presented to the Mapper and Reducer tasks. Every RecordReader is composed of the following methods:

1. `close()`
2. `getCurrentKey()`
3. `getCurrentValue()`
4. `getProgress()`
5. `initialize()`

6. `nextKeyValue()`

Processing the RDC with `FixedLength` InputFormat works until the MapReduce program encounters a record that is less than the recordLength, which typically occurs at the end of the image file.

Review of the FixedLengthInputFormat [74] and specifically the FixedLengthRecordReader code revealed the lines of code listed below from the `nextKeyValue()` method cause the MapReduce job to fail for partial records. Because Apache Hadoop is open source and uses the Apache 2.0 License we were able to take this code and modify it for our use case.

```
1  .
2  .
3  .
4  if (numBytesRead >= recordLength) {
5      if (!isCompressedInput) {
6          numRecordsRemainingInSplit--;
7      }
8  } else {
9      throw new IOException("Partial record(length = " + numBytesRead
10         + ") found at the end of split.");
11 }
12 .
13 .
14 .
```

Modifications to the above code were to remove the "else" section, lines 8 to 11, to allow the program to continue using the partial record as a valid record. In addition, line 6, which creates a byte array of size recordLength, introduces a more subtle problem that must be addressed. This byte array is initialized to zero, which is default java behavior for byte arrays.

```
1  public synchronized boolean nextKeyValue() throws IOException {
2          if (key == null) {
3                  key = new LongWritable();
4          }
5          if (value == null) {
```

```
6                   value = new BytesWritable(new byte[recordLength]);
7           }
```

However, in the `nextKeyValue()` method, this byte array is populated with contents read from the input file and the zeros are overwritten except when partial a record is encountered, in which case the record will only partially populate the byte array with valid data, leaving the rest populated with zeros. In the context of parsing for emails using a bulk_extractor, scanner this does not present a problem as it will not necessarily add or miss emails, and only adds additional processing time. However, for tasks such as the ByteCount program described in Section 4.2.3, this behavior distorts the results. The outcome is a histogram that contains significantly more zeros and overall bytes than are actually in the input files.

To resolve this problem, we add a variable, `globalSplitSize`, which is set to the actual number of bytes read not the recordLength. This variable replaces the recordLength variable in line 6 in the above excerpt. This creates and initializes a byte array to the exact size of the content, leaving no extra zeros at the end when partial records are encountered. See Appendix B for complete copy of the rawInputFormat and rawInputRecordReader with these modifications that allow MapReduce to process binary disk images stored in HDFS.

4.2.3 byteCount

The MapReduce byteCount program is modeled after the Hadoop WordCount program. Its primary purpose was to develop and test the rawInputFormat class. It was also chosen as a sample program to measure performance, as a minimal working example of an analytical program that must read and process every byte of our binary dataset. We therefore also used the byteCount tool to obtain estimates on optimal parameter settings and time required to process the Real Data Corpus (RDC).

Tuning and development of byteCount directly correlated to development of the MapReduce bulk_extractor program. The general approach of the byteCount program is to accept as input key-value pairs where value is the content of an HDFS file the size of the InputSplit. The map function receives this input, reads value contents into a byte array object then iterates over the array converting each byte to its decimal representation and incrementing the count for that byte. Meaning, if byte 65 (decimal) is encountered in the byte array, then the value at index position 65 of the Int Array of counts is increased by one.

In this program, the Int Array functions similar to a dictionary, where key is the byteValue or index position and the value is the frequency that byte occurs. A HashMap was not used due to the extreme additional memory overhead.

An important difference between byteCount and MapReduce WordCount is instead of sending each byte value and the count "one" to a Reducer, the byte value and the total count of that byte for the InputSplit is sent. This is because there are many more occurrences of bytes in a 512MiB InputSplit than words in a typical line of text in an InputSplit. Therefore, this strategy reduces the number of writes to the reducer from 536,870,912, for a 512MiB InputSplit, to 256—only one for each byte value. The Reducer receives the inputs and then processes them by summing the values for each unique byte key.

4.2.4 bulk_extractor MapReduce

Our primary goal is to implement the bulk_extractor email scanner using the MapReduce paradigm. To achieve this, we rely on the be_scan library. Be_scan is a C++ library developed at NPS that isolates bulk_extractor's scanner functionality and exposes a Java API, which allows bulk_extractor C++ libraries to be used within a Java MapReduce program. We use this tool to create bulk_extractor MapReduce, a MapReduce job similar to WordCount, except our program counts the unique email addresses found in an InputSplit and builds a histogram of email address. Once email addresses are extracted, further analysis can then be performed.

To use the be_scan library, a MapReduce job needs to load and distribute it to Mappers for execution. Prior to writing the MapReduce job, the be_scan library must be installed. (Detailed steps are available in the be_scan online documentation [76].) The installation creates the library that contains interfaces to call the bulk_extractor email scanner from Java. Once build and installation are completed, libraries located in the build directory under the `.libs` directory will need to be copied to the MapReduce project root directory to make them available during compilation of the MapReduce job. The locations where these files should be placed in this project are shown in Figure 4.2.

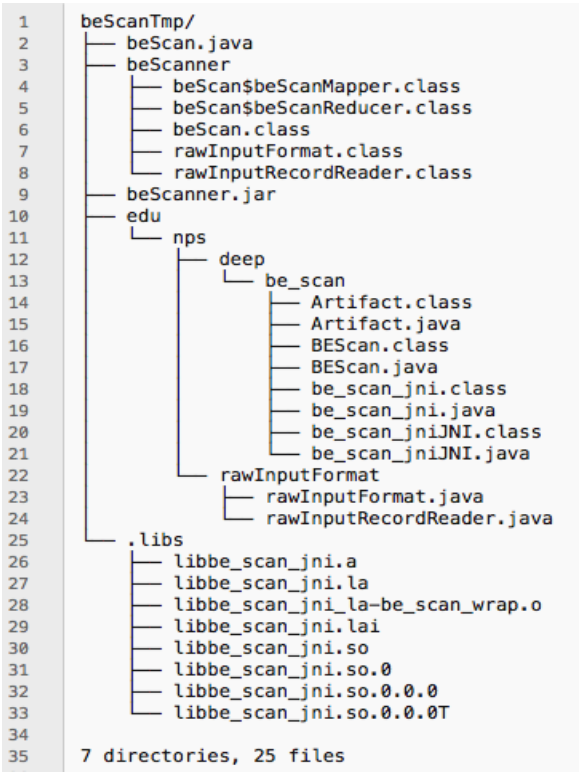

Figure 4.2. MapReduce Bulk Extractor Project Tree. Java project tree showing the `.libs` directory containing be_scan libraries. The `edu` directory contains Java program developed and be_scan. The `beScanner` directory contains class files when `beScan.java` is compiled. `beScan.java` is a MapReduce program containing user developed map and reduce functions. `beScanner.jar` is the jar file created from the contents in `edu` and `beScanner` that is used during Hadoop job submission.

In addition to the be_scan libraries, the `edu` directory tree needs to be created and populated with `be_scan` and `rawInputFormat` content as illustrated in Figure 4.2. The be_scan directory contents are from the be_scan build directory, specifically `java_bindings/edu/`. rawInputFormat contains the rawInputFormat and rawInputRecordReader found in Appendix B. The `beScanner` directory contents and `beScanner.jar` are created at compile time of the MapReduce bulk_extractor program found in Appendix E.

4.3 Experiments

This section discusses steps taken for our experiments. The four experiments are tuning Hadoop job parameters to determine suitable HDFS blocksize. Then we test throughput of converting E01 format to raw and execution of byteCount and bulk_extractor MapReduce to provide a rate if MiB/sec for each. Then, we analyze the results from byteCount and bulk_extractor MapReduce to demonstrate insights they can provide.

4.3.1 Determining an Appropriate HDFS Blocksize

The virtualized Hadoop cluster provided a platform to examine default behavior of the system when processing and storing a binary image. In the absence of processing concerns, storage of the files in HDFS is handled transparently and requires no special configuration. This is because HDFS chunks files based on the `dfs.blocksize` value in the *hdfs-default.xml* configuration file. This configuration parameter's default value is 128MiB, but is configurable at cluster install as well as when files are imported to HDFS. The significance of this is discussed below. This blocksize chunking of a file is shown in Figure 2.3. Therefore, regardless of file format, a file stored in HDFS is chunked based on block size, which correlates to the number of Mappers a MapReduce job creates.

In addition to determining that a custom InputFormat would be required, this cluster is used to determine an appropriate blocksize for the RDC in the Grace cluster. This parameter is extremely important because it affects several other settings down stream. First, a MapReduce job determines InputSplit size based on the HDFS block size [32]. InputSplits are logical divisions of the data which, by default, correspond to HDFS blocks read into memory and passed to Mappers. Specifically, InputSplit size is determined via Equation 4.1 below:

```
InputSplitSize=max(minimumSize, min(maximumSize, blockSize))    (4.1)
```

As a result, InputSplit size is generally the same as blocksize. These splits are further divided into logical records, where a record is simply a key-value pair that the map function executes on. By default, a Mapper is created for each InputSplit. Thus, the blocksize determines both the number of InputSplits thus determines how many Mappers the MapReduce job will need to execute.

InputSplit size may be customized on a per job basis, meaning a job may have larger InputSplit size without changing physical blocksize, but this is done at the expense of losing data locality. In other words, setting InputSplit size to a value that does not divide evenly into the blocksize may cause Mappers to execute on data that is stored on different dataNodes, requiring network read/writes, which degrade overall job performance. Therefore, determining an appropriate data blocksize directly affects the total number of Mappers, data local Mappers and job performance.

Our approach to narrow in on an appropriate blocksize was to store a 1TiB sample file in HDFS with four different blocksizes —128MiB, 512MiB, 1024MiB, 1536MiB and 2048MiB— then execute the byteCount MapReduce program against each of these files. Measurements that were taken follows: number of mappers, average execution time per Mapper and overall job performance time. The initial experiment was performed on the virtualized cluster but was also repeated on hardware Hadoop cluster Grace. On Grace an additional experiment was performed to measure the same items except the MapReduce bulk_extractor was used against the same 1TiB sample file. This was done to demonstrate the blocksize chosen was appropriate as well as how a different task in the Mapper affects Mapper run times.

4.3.2 Measuring Throughput

Part of measuring throughput is to measure total overhead time including dataset preparation. This preparation represents a one-time conversion cost, and might be mitigated by a process that imports images directly to HDFS. However, since many image collections are currently stored in E01 format, this preparation cost should be acknowledged. For our work, converting the RDC from E01 requires a significant amount of time. To reduce this overhead, we performed `ewfexport` experiments on a sample 8GiB disk image on Hamming changing only the number of threads `ewfexport` uses. We measure the time taken and the transfer rate for 4, 8, 16 and 32 threads to determine the optimal parameter to convert the RDC.

An additional overhead item is the time to import the converted raw disk image into HDFS. We measure this by the total time take to import a sample 1TiB RDC disk image. This value is then used to calculate and estimate to import the RDC after it has been converted. We

discuss how Hamming is used to reduce this overall time by running multiple conversions and imports in parallel. We also measured the speedup this parallelization produced.

ByteCount Throughput

We measure the success of the rawInputFormat class is done by executing a MapReduce program using it. We execute the byteCount program six times and measure the complete time taken to process the RDC and provide an average run time. This time includes setup time of all Mappers and Reducers and is collected from the Hadoop job monitoring application webpage.

Earlier implementations of byteCount ran significantly longer than the final version we developed. To diagnose performance problems we execute two standalone Java programs that mimic byteCount's Mapper. Each uses different methods of record keeping for byte values; Int Array and HashMap. We execute these two programs against a sample 512MiB disk image and measure the memory overhead of each implementation.

Bulk_Extractor Throughput

Similar to byteCount, bulk_extractor MapReduce is executed six times and we measure the complete time taken to process the RDC and provide an average run time. We compare the run time of standard bulk_extractor to the bulk_extractor MapReduce on the same 160GiB disk image to demonstrate performance gains with MapReduce.

4.3.3 Measuring Byte Frequency in the RDC

Though byteCount was designed initially to demonstrate a proof-of-concept for working with disk images on an HDFS cluster, the histogram of byte frequencies is produces provides insight into the distribution of byte values in the RDC. This information may be useful for developing baseline probabilities for analysis or anomaly detection. We analyze the histogram to explain why significant byte values may occur more frequently in the RDC.

To characterize the distribution, we use a basic histogram to plot each byte value. We further analyze the results by grouping byte values by frequency. This is plotted to further demonstrate byte values of significance in the RDC. Finally, we plot a cumulative distribution

function to show the proportion of the dataset that is composed of the most frequently occurring byte values.

4.3.4 Analysis of Email Address Distribution in the RDC

The bulk_extractor MapReduce program is designed to extract email addresses from the RDC. In addition to measuring the time taken to execute this program on the RDC we also use it to study the distribution of the email addresses it finds, a task that is difficult to do in reasonable time using conventional methods. The total number of email addresses, top email domains and top email addresses are reported and analyzed. We produce a histogram of email addresses, which we use to identify and explain the most frequently occurring addresses.

We examine whether there is a correlation between the most frequently occurring address, its higher frequency and the underlying distribution of images with respect to country of origin. We are able to determine using public methods that the most frequent email address is for a website in a specific country. We use metadata information recorded with the drive images to determine how many drives in the RDC are from a particular country.

Additionally, we calculate TF-IDF for the most frequent email address for two disk images in the RDC. This score gives insight into how strongly an email address is correlated to a particular disk image in the RDC. Finally, we examine domain name frequency and perform a comparison of the most frequent email addresses and email domains to illustrate that the most frequent email address does not belong to one of the top 10 most frequent domains.

CHAPTER 5:
Results

This chapter reports and analyzes results from our four experiments. We begin with the determination of a sufficient HDFS blocksize in Experiment 1. We discuss the E01 conversion and import timings obtained in Experiment 2, as well as the throughput of the byteCount and bulk_extractor MapReduce programs run on the Grace cluster. Next, we examine the byte frequency the RDC produced by the byteCount program in Experiment 3. The final section analyzes the distribution of email addresses in the RDC collected from Experiment 4.

5.1 Experiment 1: Determining an Appropriate HDFS Blocksize

The RDC dataset files are typically multiple GiBs with many over 100GiB. We determined 512MiB is a suitable blocksize for the RDC, and performs better than the HDFS default 128MiB blocksize. This decision is based on Apache recommendations that each map task run at least one minute, a guideline that produces the right amount of parallelism when accounting for task setup overhead [77].

Timing analysis of the byteCount program against different blocksizes, where the blocksize and recordLength are equal is shown in Table 5.1. These measurements are based on six runs for a roughly 1TiB file. Performance measurements in this table show that as blocksize and subsequently average Mapper time increases the Mapper throughput increases, which reduces the total time job execution time. A 1536MiB blocksize and record length provide the best throughput on a single 1 TiB file.

Table 5.1. ByteCount Timing for Equal Blocksize and RecordLength

BlockSize	**# of Mappers**	**Avg. Mapper Time (6 runs)**	**Avg. Mapper Throughput (6 runs)**	**Avg. Total Job Time (6 runs)**
128 MiB	7453	11sec	11.63 MiB/sec	2m 31s
512 MiB	1863	15sec	34.13 MiB/sec	1m 11s
1024 MiB	932	22sec	46.54 MiB/sec	51sec
1536 MiB	621	26sec	59.07 MiB/sec	55sec

Table 5.2. Bulk_Extractor MR Timing for Equal Blocksize and RecordLength

BlockSize	# of Mappers	Avg. Mapper Time (6 runs)	Avg. Mapper Throughput (6 runs)	Avg. Total Job Time (6 runs)
128 MiB	7453	11sec	11.63 MiB/sec	2m 29s
512 MiB	1863	16sec	32 MiB/sec	1m 12s
1024 MiB	932	25sec	40.96 MiB/sec	42sec
1536 MiB	621	24sec	64 MiB/sec	43sec

Table 5.3. Bulk_Extractor MR Timing for 1536MiB Blocksize Multiple Record Lengths

Record Length	# of Mappers	Avg. Mapper Time (6 runs)	Avg. Mapper Throughput (6 runs)	Avg. Total Job Time (6 runs)
128 MiB	7453	27sec	4.74 MiB/sec	42sec
512 MiB	1863	23sec	22.26 MiB/sec	36sec
1024 MiB	932	-	-	-
1536 MiB	621	24sec	64 MiB/sec	43sec

Tests for larger sizes were not performed because the max size of a Java `byte[]` array is 2GiB. Similar to the scenario described in Section 4.2.2, this is because data read from the InputSplit (sized record length) is stored in an array which by default is indexed with integers, which Java sets the max to 2,147,483,647 bytes. See Appendix D and E lines 37 and 49, respectively. Based on the testing performed here and the goal of finding a suitable HDFS blocksize, we chose a 512MiB blocksize as suitable for the RDC and leave increasing the record length for future work.

Similar timing analysis of the bulk_extractor MapReduce program is collected and is shown in Table 5.2. The results are the same as the byteCount program though bulk_extractor MapReduce is performing a more advanced task, extracting not just reading and counting items. Compared to byteCount results we see closely similar timings for average Mapper time of the same blocksize. Both of these tables demonstrate that the default 128MiB blocksize is not suitable for the RDC and datasets that have larger individual files, such as hundreds of GiBs.

A note to mention in both Tables 5.1 and 5.2 is that, as expected, changing the block size changes the number of mappers.

To further test our 1536MiB blocksize, we changed the record size for only bulk_extractor

MapReduce and timings are shown in Table 5.3. Two insights are gained from this table; blocksize and record have best throughput when they are equal and recordlength must be a multiple of the blocksize. A 1536MiB blocksize and record length show the best throughput even though 512MiB record length completes faster on average. The record length must be a multiple of the blocksize, which 1024MiB is not; therefore, it is not tested. Further explanation can be found later in Section 5.2.2 and Figure 5.1

This analysis on blocksize to record length performance is not exhaustive and should be explored further in future work. The optimal choice will depend on the dataset as well as the type of processing performed. Furthermore, changing the block size is time intensive because it changes the physical block storage whereas changing record size is a logical change of the data via job configuration. Our preliminary study demonstrates that 1536MiB is a reasonable choice for the blocksize; however, changing the InputSplit size directly is another way to achieve larger InputSplits and bring Mapper runtime closer to the one minute guideline.

Additional methods to achieve larger InputSplits and fewer mappers to run closer to 1 minute, such as changing InputSplit size, exist. This is based on Equation 4.1, where `minimumSize` cab be set to be greater than block size on the command line during job submissions via the following, which will effectively increase the InputSplit size:

```
-D mapreduce.input.fileinputformat.split.minsize=<value in bytes>
```

Setting this value keeps the physical blocksize the same but increases the InputSplit which increases the record(s) the map function must process. This comes at the cost of a mapper running on a data block which is not local. Therefore, this data would have to be brought to the code which conflicts with the concepts presented in big data to bring the code to the data.

This analysis of the virtualized cluster and later the Grace cluster on the effect that blocksize, record length and InputSplit have on job performance indicates that these three parameters should remain equal for best performance and parallelization. Therefore, the Real Data Corpus (RDC) is stored on Grace HDFS with both a 512MiB and 1536MiB blocksize. Though all further Hadoop jobs and timings were executed with 512MiB record size and default InputSplit size, 512MiB, determined from the block size. Though 1536MiB provides

better throughput on a single file our work and testing showed that 512MiB performed better on the RDC.

5.2 Experiment 2: Measuring Throughput

Here, we measure the throughput of three items; dataset ingest into HDFS, byteCount throughput and bulk_extractor MapReduce throughput. Specifically, we measure time taken to export E01 to raw and import into HDFS and provide timings of each. Then we time results for both byteCount and bulk_extractor MapReduce on the entire RDC.

5.2.1 Timing `Ewfexport` to Grace and HDFS

Exporting files from E01 to raw is an extremely CPU and I/O intensive task due to having to decompress the data and rewrite them to the file system. We use NPS' Hamming cluster to convert multiple E01 files at the same time on different compute nodes via the command below.

```
1 /home/tjandrze/thesisbin/bin/ewfexport -vv -q -j 8 -f raw -t /work/
      tjandrze/npsdata/$TARGET_FILE -S 0 -o 0 -B 0 $INPUT -l /work/
      tjandrze/logs/$TARGET_FILE.errors
```

Breaking down this command, the `-vv` sets the verbosity level to two to provide more information for any errors while the `-q` quiets the display of standard output. The `-f` sets the output format to raw while `-t` tells the program what the target file to convert is. The `-S`, `-o` and `-B` each specify segment file size in bytes, offset to begin and bytes to export, respectively. Each set to zero which allows for dynamic setting of the E01 segment file size and number of bytes to export and start export at byte offset zero (i.e., start from the beginning). The `-l` flag sets the path to log any errors. The most significant flag that was skipped is the `-j` flag which sets the number of concurrent processing jobs or threads to use in the export.

Performance differences between runs using 4, 8, 16 and 32 threads on a sample 8 GiB file is shown in Table 5.4. As shown, using more threads does not necessarily improve performance, likely due to resource contention. In fact, using only 8 threads has the best performance on a Hamming compute node with a transfer rate of 174 MiB/s which is far

Table 5.4. Ewfexport Performance on 8GiB File

Number of threads	Time taken	Transfer rate
4	56 seconds	146 MiB/s
8	55 seconds	149 MiB/s
16	47 seconds	174 MiB/s
32	54 seconds	151 MiB/s

below what is expected for an HPC cluster. Because extensive debugging of the HPC cluster configuration is beyond the scope of this thesis, we instead submitted SLURM jobs to convert multiple images at once to decrease the total time.

To determine time estimates of export and import into HDFS, we chose a larger file and we report those results here. The 8 GiB sample file used to determine if the number of threads improved performance is small compared to many of the files in the RDC. Therefore, we used a sample RDC file, IN8001-1000.E01, which converts to file IN8001-1000.E01_1469874896.raw. This file is roughly 1TiB in size and takes roughly 81 minutes to export to raw format. Roughly, this means it would take 9,396 minutes to convert the complete 116 TiB RDC to raw sequentially. Once converted this sample RDC file is put into HDFS on Grace using the Hadoop command below, which takes an additional 185 minutes. Running sequentially on the entire RDC would take approximately 23,578 minutes or 16 days. The estimated total time to convert and import the RDC sequentially is therefore 32,974 minutes or 22 days. Using Hamming to convert and import up to five images at a time is estimated to take a fifth of the time or 6,594 minutes.

The authors recognize this amount of time to convert to raw is a significant amount of preprocessing time; therefore, results from this thesis work best with files already in raw format or for analysts who have time and resources to convert larger data sets to raw format.

5.2.2 Throughput of ByteCount and Bulk_Extractor MapReduce

Using the rawInputFormat in the byteCount program takes an average of 1 hour 48 minutes (over six runs) to process the RDC's 116TiB using a 512MiB record Length and InputSplit. This execution created 240,313 map tasks with 235,122 of those being data local. This run time creates an average throughput of 18,770 MiB/sec as shown in Table 5.5. Similar test for bulk_extractor MapReduce completed in an average of 2 hours 5 minutes on six runs to process the RDC. Therefore the throughput of the bulk_extractor MapReduce is 16,217

MiB/sec, is shown in Table 5.5.

To compare these results, we attempted to run the standard bulk_extractor on a sample disk image. The standard bulk_extractor we use is turning off all other scanners except the email one. A 160GiB is used for a direct comparison. Using bulk_extractor MapReduce, the job takes 34 seconds while the standard bulk_extractor on the same file takes 30 minutes using 24 cores.

Table 5.5. ByteCount and Bulk_Extractor Throughput

Job Type	**Avg. Time taken(6 runs)**	**Throughput**
byteCount	1h 48mins	18,770 MiB/sec
bulk_extractor MapReduce	2h 5mins	16,217 MiB/sec

The throughput and execution times of both byteCount and bulk_extractor demonstrate that using MapReduce significantly outperforms over traditional digital forensics techniques.

The rawInputFormat was developed as the foundation for both byteCount and bulk extractor. Its primary function is to read the binary input files and create records for the map function. Recall Figure 2.3 which illustrates how the physical blocks of a raw disk image are stored in HDFS. When these blocks are read by a Hadoop job, rawInputFormat determines the logical division of the data.

We illustrate in Figure 5.1 that rawInputFormat logically divides the blocksize of 512 MiB further into an InputSplit and then to a Record, which is then passed to a Mapper. Following the results of Experiment 1, we set blocksize, InputSplit, and record Length to the same value, but these values may be changed, as described in Section 5.1. The distribution of files in HDFS to a processing Mapper is controlled by rawInputFormat and specifically by setting the InputFormat class during Hadoop job configuration as seen below:

```
job.setInputFormatClass(rawInputFormat.class);
```

Something learned, but not necessarily a result, from the byteCount program that is implemented in the MapReduce bulk_extractor program is the method to keep track of bytes. It serves as a reminder that exceeding memory limitations is very easy to do. A common

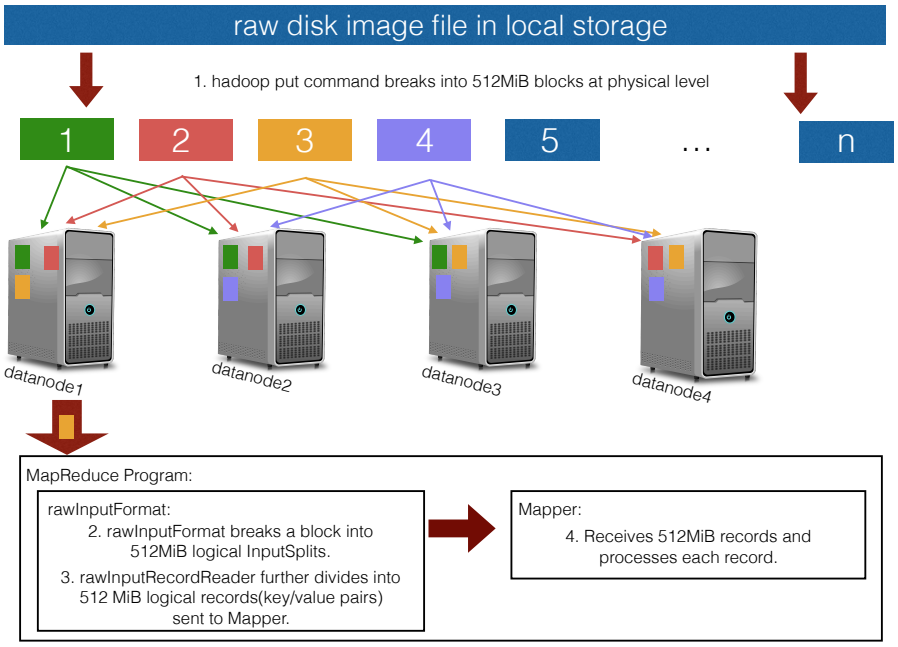

Figure 5.1. File Representation to a Mapper. A file is physically divided into 512MiB blocks by the `hadoop fs -put` command. Blocks are stored on datanodes which are read by a MapReduce program. MapReduce program specifies rawInputFormat is used which logically divides the block into InputSplits which the rawInputRecordReader divides into records sent to the Mapper for processing.

approach is to use a dictionary type data structure where the byte value is the key and the frequency is the value. Java's implementation of a dictionary data structure is the HashMap [78]. The HashMap implementation for Integers creates large number of Integer objects which require a minimum of 16 bytes of memory.

For a 512MiB sample file, which reflects the InputSplit or Mapper size of the MapReduce byteCount program, this equates to roughly 318 MiB of memory or 38% of the total memory used by the program. We illustrate in Figure 5.2 the Integer object is the second largest memory consumer of the program, behind the object holding the actual data. Using a HashMap object in the MapReduce program each mapper requires 38% more memory, or approximately 62% the size of the InputSplit. This is a significant amount of overhead if

Class Name	Instances [%]	Instances	Size
byte[]		1,186 (10.3%)	537,062,599 (99.4%)
int[]		526 (4.6%)	2,663,560 (0.5%)
char[]		3,240 (28.3%)	282,444 (0.1%)
java.lang.String		2,598 (22.7%)	72,744 (0%)
java.lang.Object[]		580 (5.1%)	51,944 (0%)
java.nio.HeapCharBuffer		778 (6.8%)	41,234 (0%)
java.lang.String[]		106 (0.9%)	15,496 (0%)
java.lang.reflect.Field		122 (1.1%)	13,786 (0%)
java.util.HashMap$Node		251 (2.2%)	11,044 (0%)
java.lang.StringBuilder		382 (3.3%)	10,696 (0%)
java.util.HashMap$Node[]		20 (0.2%)	10,464 (0%)

Figure 5.3. Memory Profile of Int Array. This figure is sorted by size (far right column). That Java `byte[]` object consumes 99% of the total memory for the program. The second largest memory consumer is `int[]` which is used for indexing the array, but is significantly less, 0.5%, compared to 38% used by the HashMap memory in Figure 5.2.

the goal is not necessarily fast lookup times. Across the 1,920 Mappers on the Grace cluster this amounts to nearly 600GiB of additional memory overhead.

Class Name	Instances [%]	Instances	Size
byte[]		411 (0%)	536,994,650 (61.3%)
java.lang.Integer		16,695,597 (100%)	333,911,940 (38.1%)
int[]		190 (0%)	4,538,248 (0.5%)
char[]		1,775 (0%)	136,126 (0%)
java.lang.Object[]		558 (0%)	49,320 (0%)
java.lang.String		1,753 (0%)	49,084 (0%)
java.util.HashMap$Node		507 (0%)	22,308 (0%)
java.util.HashMap$Node[]		15 (0%)	10,344 (0%)

Figure 5.2. Memory Profile of HashMap. This figure is sorted by size (far right column). This is the size in memory each class or Java object is using. With over 16 million instances the Java `Integer` object used for record keeping in a HashMap object takes up 38% of total memory. The `byte[]` object is the object which stores the contents of the record read from file, and therefore is expected to be the largest memory consumer.

An alternative to Java HashMap is to use a lower level implementation of a dictionary data structure. That is to create an Int Array of length 256. The index position of the array is the key and the element at that index position is the value. This approach reduces total memory for a 512MiB file from 875,810,190 bytes to 540,337,183 bytes or roughly the memory footprint of the Integer objects in the HashMap implementation. This reduction is illustrated in Figure 5.3 and demonstrates the actual data read in consumes 99% of the memory as expected.

The effectiveness of rawInputFormat is best represented by the successful execution of

byteCount and bulk_extractor since this demonstrates successful execution of a Hadoop job on binary images in HDFS. Therefore, the next two subsections analyze results and successful execution of byteCount and bulk extractor on the RDC.

5.3 Experiment 3: Byte Frequency in the RDC

The byteCount program provides several insights into the operation of MapReduce programs using rawInputFormat. Its output is a histogram of unique bytes in the RDC. In addition to generating a histogram, this MapReduce program helps illuminate details of memory usage and limitations of two Java objects, IntArray and HashMap. A third achievement from the byteCount program is the actual frequency results from executing the byteCount program. Appendix F contains two complete tables of these results, one sorted by byte values and the other by byte frequency.

A basic histogram of the results graphed using log scale for the yaxis is shown in Figure 5.4. This figure illustrates that there are some byte values with significantly higher frequencies than others. Knowing these bytes and understanding the difference may provide insight useful for detecting anomalies or making predictions about data.

To better clarify the trends in the data, we present a different representation of the byteCount results in Figure 5.6. This figure groups the byte values based on the frequency that byte occurs. As figure makes clear, in the Real Data Corpus 132 of the 256 possible byte values have frequencies that fall between 200 and 300 billion. Notably only six byte values occur more than 500 billion times in the RDC with three of those occurring more than one trillion times. Those byte values are 48, 1, 32, 0, 246, and 255. Combined, they represent over half of the total space. This is illustrated by the Cumulative Distribution Function shown in Figure 5.5. A Cumulative Distribution Function of the byte frequencies can be found in Figure 5.5 illustrating that over half of the RDC is empty space. The impact these last six values and in particular the byte value 0 has in the RDC is shown in Figure 5.5 and Table 5.6. These illustrations show that 48.2% of the RDC is any other byte value besides 0 or 255.

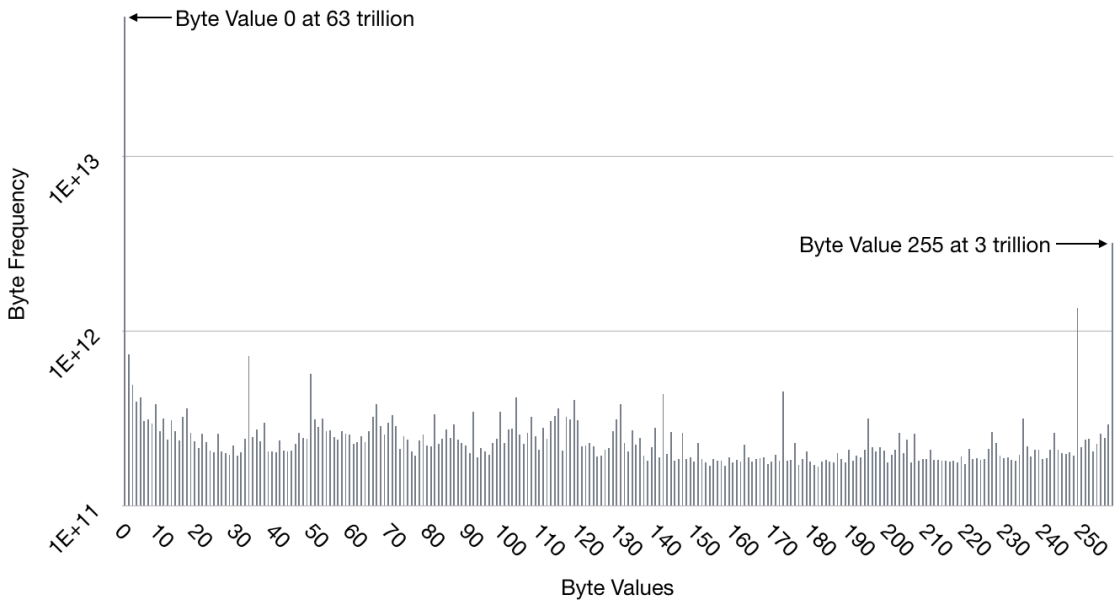

Figure 5.4. Byte Count Results Histogram Log Scale. Every byte value, 0-255, plotted on a log scale. The far left line is the count for byte value 0 while the far right is byte value 255 with several spikes in between corresponding to byte values 1, 32, 48 and 246 all above 500 billion

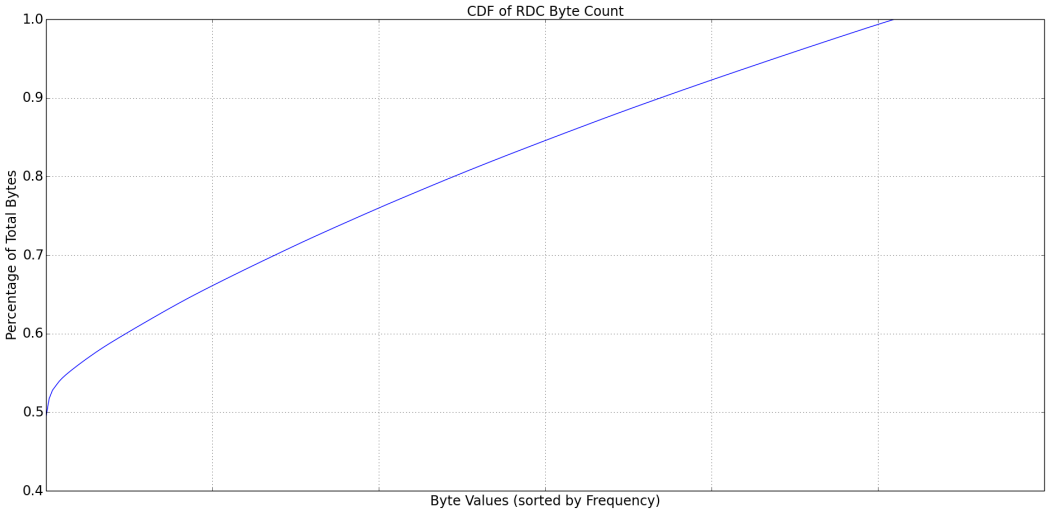

Figure 5.5. Byte Count Cumulative Distribution Function. From left to right along the x-axis are byteCount values sorted by frequency with their respective cumulative percentage along the y-axis. The first values along the x-axis are 0, 255 and 246 are the order of byte found in Table 5.6. A complete order of these values can be found in Appendix F.

Table 5.6. Highest Frequency Bytes Percentage

Byte Value	Byte Frequency	% of RDC
48	570,435,823,506	0.44%
32	716,338,085,559	0.55%
1	734,295,502,830	0.57%
246	1,351,928,628,522	1.05%
255	3,183,095,937,209	2.47%
0	63,273,879,033,072	49.27%

Values of the highest frequency bytes in the RDC are shown in Table 5.6. These values occur more frequently and the authors theorize this is based on common default values representing empty space. That is, prior to any writing to disk these spaces on the hard drive are initialized to some value. Any non-empty space means a deliberate change occurred (i.e., a write of data). This empty space can be thought of as the background of a picture, defaulting to some pre designated color, and the non-empty space in the foreground reflects some positive change such as adding subjects to the picture.

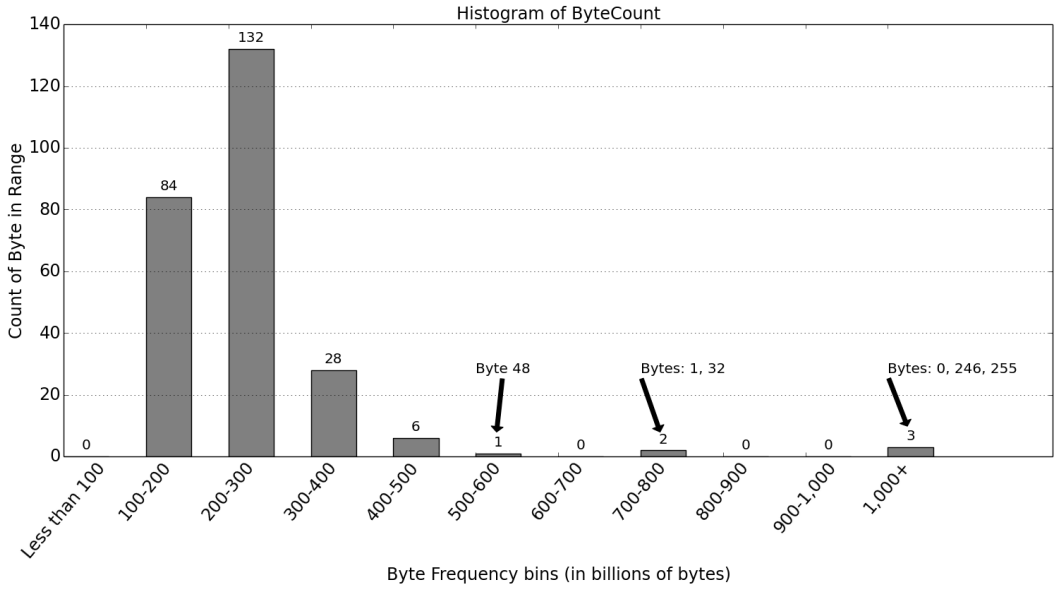

Figure 5.6. Byte Count Grouped Results. ByteCount histogram of bytes grouped into 100 billion frequency bins. Arrows annotate that bytes 48, 1, 32, 246, 255 and 0 are the only byte values to occur more than 500 billion times in the RDC. While the majority of byte values occur between 200 and 300 billion times.

The highest occurring byte values by far are 0 and 255. From the default empty space point of view these values are expected as 0 is the first and easiest initialization value while 255, all 1s in binary representation, would be the inverted initialization. These two values also are the most frequently used values in formatting and wiping hard drives, where the hard drive is completely overwritten with these values. The next most frequent byte value is 246. The hexadecimal representation of this is `0xF6`. This hexadecimal value is a filler value that older implementation of the File Allocation Table (FAT) file system uses in the data region for unused parts. Additionally this hexadecimal value is also used in many disk overwriting tools to overwrite disk data with `0xF6` instead of `0x00` or `0xFF`. The byte value of 1 is the next most common, which is the simplest non-zero initialization value used. The byte value 32, which represents the "space" character in ASCII, is used frequently in documents. Byte

value 48 represents the ascii character zero.

A Cumulative Distribution Function of the byte frequencies can be found in Figure 5.5 illustrating that over half of the RDC is empty space. The impact these last six values and in particular the byte value 0 has in the RDC is shown in Figure 5.5 and Table 5.6. it can be seen in these tables that 48.2% of the RDC is any other byte value besides 0 or 255.

5.4 Experiment 4: Analysis of Email Address Distribution in the RDC

Executing bulk_extractor MapReduce against the RDC extracts 223,332,658 total email addresses of which 12,882,638 are unique and 12,673,155 occurring less than 100 times. The most frequent email address, by a large margin, extracted from the RDC is "janglo@yahoogroups.com" which occurs 2,155,155 times. Email addresses that occur more than 500,000 times in the RDC are shown in Table 5.7.

Seven out of the 12,882,638 unique email addresses occurred more than one million times, while 4,580,427 addresses occurred only once which is shown in Figure 5.7. Those seven email addresses are listed in Table 5.7. Additionally, seven of the top 15 email addresses that appear over 500,000 times the authors deem are personal email addresses. This is an anomaly that might be investigated further to determine if these addresses correspond to malicious users or persons of interest in an investigation. An additional anomaly in Figure 5.7 is the significantly higher number of email addresses occurring between 101 and 1,000 times. This may be because this range contains 899 frequency values due to the way we binned the results, but may be worth further investigation.

For instance, the top email address "janglo@yahoogroups.com" is an online community for Israel's English speakers and in 2007 it was one of the most active Yahoo Groups [79]. The fact this email address occurs significantly more in the RDC may be attributed to there being 288 of the 3,088 hard drives in the RDC beginning with IL or Israel's country code. The IL country code drives are the third most frequent in the RDC behind only China (CN) and India (IN) with 745 and 667, respectively as illustrated by Table 5.8. Furthermore, these 288 drives account for 36.5TiB out of the total 116TiB. In contrast, the 745 CN drives in the RDC only account for 1.3TiB of the RDC and IN drives only 9.9TiB.

Table 5.7. Real Data Corpus Top Email Addresses

Domain	Frequency
janglo@yahoogroups.com	2,155,155
CPS-requests@verisign.com	1,392,629
singmed@singmet.com.sg	1,390,464
personal email address 1 [redacted]	1,289,389
marketing@pacific-conferences.com	1,241,982
janglo-unsubscribe@yahoogroups.com	1,135,928
personal email address 2 [redacted]	1,016,310
list@shemesh.co.il	803,363
personal email address 3 [redacted]	670,526
premium-server@thawte.com	666,195
ubuntu-devel-discuss@lists.ubuntu.com	653,370
personal email address 4 [redacted]	652,213
personal email address 5 [redacted]	618,299
personal email address 6 [redacted]	601,005
personal email address 7 [redacted]	521,085

Different frequency groups in which email addresses occurred is illustrated in Figure 5.7. The majority of email addresses extracted occurred fewer than 100 times. These results in the hands of a forensic analyst provide direction to investigate the significance of each of those email addresses that occur frequently in 116TiB Real Data Corpus. Applications of these results include: triage, probabilistic whitelisting and anomaly detection.

Table 5.8. Real Data Corpus Country Codes

Country Code	Frequency
CN	745
IN	667
IL	288
SG	225
TH	188
MX	171
PS	139
AE	87
PK	84
MY	78
BD	57
CA	53
AT	44
DE	41
BS	34

Country Code	Frequency
UK	26
RS	24
CZ	22
HU	22
GH	20
PA	17
MA	11
TR	10
HK	8
BA	7
EG	7
GR	7
JP	4
CH	2

To further investigate the significance of the relationship between an email address in the

Real Data Corpus(RDC) and to a specific drive, we can calculate its Term Frequency-Inverse Document Frequency (TF-IDF) value. Using “janglo@yahoogroups.com” and hard drives “IL008-0003.E01_1469968220.raw” and “IL009-0004.E01_1469964873.raw” as examples, we find the TF-IDF values indicate this email is strongly correlated with the “IL009-0004.E01_1469964873.raw” image and much less with the “IL008-0003.E01_1469968220.raw” image.

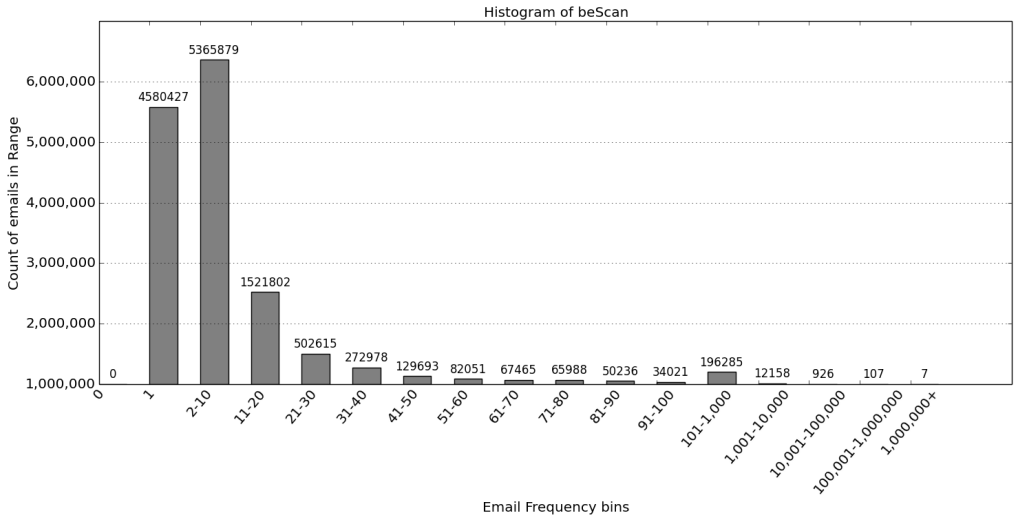

Figure 5.7. Bulk Extractor Grouped Results. The majority of email addresses found in the RDC occur less than 100 times while only seven email addresses occur more than one million times. The trend of fewer emails occurring more frequently is expected, but the jump for emails occurring between 101 and 1,000 times may be worth further study.

The TF-IDF score of the “IL008-0003.E01_1469968220.raw” drive is 0.00003258 while the TF-IDF of the “IL009-0004.E01_1469964873.raw” drive is 0.257225. The email address appears on 59 drives, but only once out of the 52,751 email addresses in “IL008-0003.E01_1469968220.raw,” as opposed to 1,551,910 times in “IL009-0004.E01_1469964873.raw.” The significance of these scores to an analyst is they may start their investigation of the email address, “janglo@yahoogroups.com,” with the “IL009-0004.E01_1469964873.raw" drive, but we already know it is not very interesting because

it occurs too many times. However, if we have an email from this address to another email address the TF-IDF of the other email address will likely outweigh the TF-IDF of "janglo@yahoogroups.com." This insight may assist an analyst prioritize drives to examine.

The above calculations illustrate that "janglo@yahoogroups.com" is much more significant in IL009-0004.E01_1469964873.raw. A forensic analyst can take the frequencies from this MapReduce bulk extractor program and calculate TF-IDF on an email address that is already prevalent in an investigation to determine potential hard drives to examine further. Or the analyst may calculate the TF-IDF of every email address extracted for every drive in the investigation. Appendix G contains a sample python program to calculate TF-IDF values based on the results from this MapReduce bulk extractor program.

In addition to specific email address frequency in the RDC, the distribution of email domains are also items which may provide analysts further insight into a disk image during an investigation. The two most frequent email domains in the RDC are "gmail.com" and "hotmail.com" with 612,900 and 681,258 occurrences, respectively, while only 11 domains occurred more than 100,000 times. Table 5.9 contains a listing of the domains with over 100,000 occurrences in the Real Data Corpus. This table does not count duplicate email addresses in the same domain. That is, the 2,155,155 occurrences of "janglo@yahoogroups.com" are counted as one occurrence for the "yahoogroups.com" domain.

Table 5.9. Real Data Corpus Top Unique Email Domains

Domain	**Frequency**
hotmail.com	681,258
gmail.com	612,900
yahoo.com	492,479
capgemini.com	281,564
db.com	270,220
francenet.fr	250,790
aol.com	184,269
aig.com	157,046
yahoo.co.in	146,747
AIG.com	109,518
corp.capgemini.com	102,930

Extracting email addresses in an average of 2h 5mins from a large dataset such as the RDC puts results into an analyst's hands quickly. In addition, the collection-scale statistics

that MapReduce bulk_extractor makes available can contribute to better decision-making and automate triage capabilities, including dynamic creation of whitelists containing email addresses that are not relevant or extremely relevant to an investigation depending on the scenario. The MapReduce program thus provides enhanced ability to extract emails and quickly get pertinent results to the analyst for further investigation at a scale that could take days to weeks to process using traditional methods.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusions and Future Work

In this final chapter, we review the goals, results and contributions of our research. We look back at and respond the research questions presented in Chapter 1. After summarizing our results and contributions, we close with some recommendations on future research on this topic.

6.1 Conclusions

Our primary motivation was to provide a distributed tool for forensic analysts to reduce monetary costs, time and specialized knowledge required for a forensic investigation of a large dataset. To achieve this goal, we develop a Hadoop InputFormat class capable of handling raw disk images and we use this InputFormat class to implement bulk_extractor MapReduce, a massively parallel email address extraction tool using the MapReduce paradigm. During the process of achieving these goals, we also develop an additional forensic tool, a MapReduce program to count bytes in the Real Data Corpus, which may be used to determine baseline probabilities in future research.

MapReduce provides an inherently distributed foundation that hides many of the complexities required in other distributed processing methods. The hiding of traditional distributed computing complexities greatly reduces monetary and knowledge costs often associated with scaling. Traditional digital forensics and distributed processing tools require very specialized systems and software, which may unnecessarily burden an analyst performing the complex digital forensic analysis often required. Hadoop and MapReduce can help to reduce this burden, as well as reducing infrastructure costs by running on cost-effective commodity hardware.

This thesis made the following contributions:

1. We perform exploratory analysis of the feasibility of using a Hadoop cluster and HDFS to store raw disk images and study block level parameter impacts on MapReduce jobs.
2. We provide a method for conversion of the Real Data Corpus from a E01 format to a

raw format for storage in HDFS in a practical timeframe.

3. We develop a new InputFormat for processing raw disk images stored in HDFS with an average throughput of 18,770 MiB/sec and 16,217 MiB/sec on the RDC for byteCount and bulk_extractor, respectively.
4. We developed a MapReduce byteCount program capable of analyzing 116TiB dataset in an average time of 1h 41mins. This new tool may be useful for developing improved triage and anomaly detection tools in the future.
5. We developed a MapReduce bulk_extractor email scan program capable of analyzing 116TiB dataset in an average of 2h 5mins. We therefore provide a quick, cost-effective distributed tool that can directly aid a forensic investigation.

In light of the above results, we recommend MapReduce clusters as a viable solution to scale digital forensics tools. Our work demonstrates the ability to analyze the 116TiB Real Data Corpus in 2h 5mins on a 25-node Hadoop cluster. This result suggests that our approach will scale with growing datasets. Additionally, our system satisfies five out of the six requirements for distributed digital forensics (DDF) defined in Chapter 3. We revisit these requirements below and evaluate our work with respect to each:

1. *Scalable.* The MapReduce paradigm allows for the quick and easy addition of new datanodes without impact to currently running cluster. Our work did not focus on measuring whether scaling out Hadoop provided near linear improvements, but several related works mentioned in Chapter 3 discuss these improvements. They conclude that scaling out Hadoop provides improvement, but the scale of improvement is dependent on each Application. Our work demonstrates a major performance improvement over traditional methods showing that a non-distributed bulk extractor using 24 nodes completed in 30 minutes compared to 34 seconds utilizing bulk_extractor MapReduce.
2. *Platform-independent.* Hadoop and MapReduce are designed to run on commodity and spare hardware. The only requirements are the operating system must be a compatible Linux operating system and have the same version of Java installed on all nodes. We argue that our tools and results could be made available through a web interface.
3. *Lightweight.* Installation of a Hadoop cluster has minimal requirements and steps. Installation of a fully-distributed cluster requires assigning environment variables and configuring a minimum of 4 files up to 6 files depending on the environment. Once

these files are configured the final step is to format the HDFS then start Hadoop services. For this research, installation of Hadoop 2.6 on a 6 node virtualized cluster took a few hours until the cluster was capable of running provided example MapReduce jobs. We acknowledge that purchasing time and physical hardware setup will add to this time. With its minimal node requirements and short turn around to execution, this makes MapReduce a very lightweight solution compared to many of the existing attempts to distribute digital forensics.

4. *Interactive.* In the example we developed, interacting with the results during job execution is not possible. This is because the method and order in which Mappers and ultimately Reducers execute. Specifically, Reducer tasks do not begin executing until 80% of map tasks are complete. Therefore, there are no results to interact with until Reducers begin executing. This value is configurable, but the authors did not explore the impact to performance if this is increased or decreased.
5. *Extensible.* Adding a new function or MapReduce job requires a developer to write two functions, a mapper and reducer. Additionally, creating a custom input format, such as rawInputFormat in this thesis, requires two files.
6. *Robust.* Default configuration of a Hadoop cluster is to have three block replicas at all times. These replicas allow for continued data availability if a datanode fails. Additionally, these replicas allow for a higher level of parallelism to ensure one datanode does not become a bottle neck during job execution. This means that if a node fails the MapReduce job tracker detects this via failed heartbeat responses and automatically spawns a new Mapper or Reducer using one of the block replicas. Block replicas provide analysts with the confidence that if a node fails data on its hard drives is not lost.

Finally, we review and provide answers to out motivating questions from Chapter 1:

1. Can the MapReduce paradigm be leveraged to provide a distributed computing method to reduce digital forensic tool execution time and cost?

   Yes. We have shown that using MapReduce provides significant performance gains in terms of reduce time and cost. The time to execute on a sample 160GiB file dropped to under one minute compared to 30 minutes using traditional methods. Reduced execution time directly contributes to reduced cost. In addition, we argue that our approach will lead to reduced tool development costs by avoiding complex

programming models needed to achieve similar performance benefits using traditional HPC. Additionally, MapReduce and Apache Hadoop software is open-source with no annual fees for usage and upgrades, and can be readily modified to perform forensics analysis.

2. What best practices should be used to implement a MapReduce approach to Digital Forensics?

   The first best practice is to select an appropriate blocksize prior to storage of data in HDFS. This blocksize directly contributes to number of Mappers and how many of those are datanode local, which are key items to achieve optimal parallelization. We recommend a 512MiB blocksize for the RDC.

   A second best practice is to be aware of what the Mapper writes to a Reducer. This process writes to a temporary local disk file which the Reducer will read from. Therefore, reducing the amount of times and data a Mapper must write can greatly reduce the program execution time. We recommend implementing a combiner function, similar to that used by the byteCount program, within the Mapper to reduce these writes.

   A third best practice is to carefully monitor memory requirements of each aspect of a MapReduce job. The Application Manager, Mapper and Reducer each have different parameters for tuning memory. To set these correctly, it is important to understand that container memory allocation is not the actual Java process memory available. Rather, it is actually less because of the container's own memory requirements. Moreover, one must understand that the Java object usage in a MapReduce program affects what these parameters should be if they need to change from the default.

3. Is the MapReduce solution to digital forensics enough to keep up with growing digital forensics data volumes?

   Yes. MapReduce is a solution to the growing digital forensics volume crisis. MapReduce is open-source which greatly lowers the cost compared to other distributed solutions. Additionally, MapReduce is designed to work on commodity hardware which is less expensive. Many analysts may already have intermediate programming knowledge which is easily transferable to MapReduce programs. MapReduce's essential difference from traditional distributed computing is its strategy of bringing the code to the data. As disk image size increases and more devices are included in an investigation this strategy is increasingly necessary, since moving this data set around

becomes more and more of a burden.

6.2 Future Work

While our research provides proof-of-concept tools to conduct a byte count and extract email address features on a 116TiB data corpus within 2h 5m, there is much work that remains before this approach can be integrated into a production analysis system. Some areas for improvement include creation of additional bulk extractor scanners, further tuning of blocksize, InputSplit size and record length parameters, adaptation of other digital forensics tools to utilize rawInputFormat. In addition, future work might explore creation of an InputFormat to work with E01 file format. Progress made in these areas will further push digital forensics tools into the distributed processing paradigm that is needed to address growing data volume challenges.

Our work provides basic analysis of some elementary features of a disk image by counting unique bytes and extracting email addresses. Future work could build on these features to develop higher-level analytics. For example, cross drive analysis should be explored to determine correlation between drives. Features acquired from other bulk_extractor scanners, such as URLs and credit card numbers, could contribute to this analysis. In addition, incorporating byte offsets of the artifacts could provide a detailed starting point for developing analysis. High-level analytical tools utilizing MapReduce should benefit from performance gains and collection-scale processing capabilities.

The goal of this work is to determine feasibility of MapReduce as a viable solution to the growing data volume challenge. Therefore minimal time was spent determining optimal tuning parameters for MapReduce programs, though we perform some preliminary work in this direction. Therefore, improvements may be made from using additional optimization of Hadoop and MapReduce parameters such as blocksize, InputSplit size, record length and memory usage.

Bulk_extractor is a digital forensic tool that is capable of extracting many features, but there are an abundance of additional tools that cover different aspects of an investigation and could also benefit from a massively parallel approach. MapReduce implementation of other tools will advance the start-of-the-art by increasing the number of tools capable of analyzing larger datasets and performing large cross drive analysis. These tools could

utilize the rawInputFormat developed in this research.

A final area of improvement is to develop an InputFormat class capable of processing E01 file format. The majority of the forensic community is familiar with and uses this format, therefore the capability to run the bulk_extractor email scanner from this research could be improved by requiring less storage space in HDFS. Decreased storage space means less costs associated with a Hadoop cluster. In addition to an E01 format InputFormat class, future work into quicker methods to import data into HDFS should be explored to reduce pre-processing time.

APPENDIX A: Converting E01 to Raw

NPS's Real Data Corpus is stored in E01 format. This format works to save storage space, but adds additional complexities for our research. Therefore we spent time up front converting all files in the RDC to raw format prior to import into HDFS. We realize this requires a large amount of pre-processing and as such we utilized NPS's Hamming cluster. Below are bash job scripts we utilized to achieve this task. We also note that this method of converting is unique to NPS's HPC environment, which has a shared parallel file system mounted to both Hamming and Grace clusters.

A.1 e01ConvertSlurm.sh

To convert E01 to raw the libewf library is required [13]. After conversion, the RDC size is 128 TiB stored in HDFS.

```
1  #!/bin/bash
2  #
3  #SBATCH --nodes=1
4  #SBATCH --ntasks-per-node=16
5  #SBATCH --time=24:00:00
6  #SBATCH --mem-per-cpu=10gb
7  #SBATCH --output=/home/tjandrze/outputs/array_%A_%a.out
8  #SBATCH --error=/home/tjandrze/errors/array_%A_%a.error
9  #SBATCH --array=1-3089%5
10
11 #####   array=1-TotalNumberOfFiles%5
12
13  hostname; date
14
15 #filelist created via find /work/DEEP/corpus/nus/drives/ -type f -
       size +0c -name "*.E01" >filelist
16 INPUT=$(sed -n "$SLURM_ARRAY_TASK_ID"p /home/tjandrze/filelist)
17 echo $INPUT
18 TARGET_FILE=$(basename $INPUT)_$(date +%s)
19 echo $TARGET_FILE
```

```
20
21 /home/tjandrze/thesisbin/bin/ewfexport  -vv -q -j 8 -f raw -t /work/
       tjandrze/npsdata/$TARGET_FILE -S 0 -o 0 -B 0 $INPUT -l /work/
       tjandrze/logs/$TARGET_FILE.errors
22 wait
23 echo $INPUT >>/work/tjandrze/logs/TARGET_FILE.log
24
25 ssh $(host grace|head -1|awk '{print $NF}') "/home/tjandrze/hdfsCopy
       .sh /work/tjandrze/npsdata/$TARGET_FILE.raw"
26 wait
```

A.2 hdfsCopy.sh

```
1 #!/bin/bash
2 /usr/bin/hdfs dfs -put $1 /user/tjandrze/DEEP/input/
3 wait
4 /bin/rm -f $1
5 wait
6 /bin/rm -f $1.info
```

APPENDIX B: rawInputFormat Class

This appendix contains the source code of the two Java files written to allow MapReduce jobs to process binary disk images stored in HDFS.

B.1 rawInputFormat.java

```
1  /**
2  * This code was modified from the original Apache Hadoop
       FixedLengthInputFormat.java
3  * code. As such, a copy of the Apache License, Version 2.0 may be
       obtained at
4  *
5  *                        https://www.apache.org/licenses/LICENSE-2.0.
       html
6  *
7  *
8  *
9  */
10
11 import java.io.IOException;
12 import java.io.InputStream;
13
14 import org.apache.hadoop.classification.InterfaceAudience;
15 import org.apache.hadoop.classification.InterfaceStability;
16 import org.apache.hadoop.conf.Configuration;
17 import org.apache.hadoop.fs.FSDataInputStream;
18 import org.apache.hadoop.fs.FileSystem;
19 import org.apache.hadoop.fs.Path;
20 import org.apache.hadoop.fs.Seekable;
21 import org.apache.hadoop.io.BytesWritable;
22 import org.apache.hadoop.io.LongWritable;
23 import org.apache.hadoop.io.compress.CodecPool;
24 import org.apache.hadoop.io.compress.CompressionCodec;
25 import org.apache.hadoop.io.compress.CompressionCodecFactory;
26 import org.apache.hadoop.io.compress.CompressionInputStream;
```

```
27 import org.apache.hadoop.io.compress.Decompressor;
28 import org.apache.hadoop.mapreduce.InputSplit;
29 import org.apache.hadoop.mapreduce.JobContext;
30 import org.apache.hadoop.mapreduce.RecordReader;
31 import org.apache.hadoop.mapreduce.TaskAttemptContext;
32 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
33 import org.apache.hadoop.mapreduce.lib.input.LineRecordReader;
34 import org.apache.commons.logging.LogFactory;
35 import org.apache.commons.logging.Log;
36
37 public class rawInputFormat extends FileInputFormat<LongWritable,
      BytesWritable> {
38       public static final String FIXED_RECORD_LENGTH = "
            fixedlengthinputformat.record.length";
39
40       public static void setRecordLength(Configuration conf, int
            recordLength) {
41             conf.setInt(FIXED_RECORD_LENGTH, recordLength);
42       }
43
44       public static int getRecordLength(Configuration conf) {
45             return conf.getInt(FIXED_RECORD_LENGTH, 0);
46       }
47
48       @Override
49       public RecordReader<LongWritable, BytesWritable>
            createRecordReader(InputSplit split, TaskAttemptContext
            context) throws IOException, InterruptedException {
50             int recordLength = getRecordLength(context.
                  getConfiguration());
51             if (recordLength <= 0) {
52                   throw new IOException("Fixed record length "
                        + recordLength + " is invalid. It should
                        be set to a value greater than zero");
53             }
54             return new rawInputRecordReader(recordLength);
55       }
56
57       @Override
```

```
58          protected boolean isSplitable(JobContext context, Path file)
                {
59                  final CompressionCodec codec = new
                        CompressionCodecFactory(context.getConfiguration
                        ()).getCodec(file);
60                  return (null == codec);
61          }
62  }
```

B.2 rawInputRecordReader.java

```
1  /**
2  * This code was modified from the original Apache Hadoop
       FixedLengthInputFormat.java
3  * code. As such, a copy of the Apache License, Version 2.0 may be
       obtained at
4  *
5  *                        https://www.apache.org/licenses/LICENSE-2.0.
       html
6  *
7  *
8  *
9  */
10
11 import java.io.IOException;
12 import java.io.InputStream;
13
14 import org.apache.hadoop.classification.InterfaceAudience;
15 import org.apache.hadoop.classification.InterfaceStability;
16 import org.apache.hadoop.conf.Configuration;
17 import org.apache.hadoop.fs.FSDataInputStream;
18 import org.apache.hadoop.fs.FileSystem;
19 import org.apache.hadoop.fs.Path;
20 import org.apache.hadoop.fs.Seekable;
21 import org.apache.hadoop.io.BytesWritable;
22 import org.apache.hadoop.io.LongWritable;
23 import org.apache.hadoop.io.compress.CodecPool;
24 import org.apache.hadoop.io.compress.CompressionCodec;
25 import org.apache.hadoop.io.compress.CompressionCodecFactory;
26 import org.apache.hadoop.io.compress.CompressionInputStream;
```

```
27 import org.apache.hadoop.io.compress.Decompressor;
28 import org.apache.hadoop.mapreduce.InputSplit;
29 import org.apache.hadoop.mapreduce.lib.input.FileSplit;
30 import org.apache.hadoop.mapreduce.JobContext;
31 import org.apache.hadoop.mapreduce.RecordReader;
32 import org.apache.hadoop.mapreduce.TaskAttemptContext;
33 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
34 import org.apache.hadoop.mapreduce.lib.input.LineRecordReader;
35 import org.apache.commons.logging.LogFactory;
36 import org.apache.commons.logging.Log;
37
38 public class rawInputRecordReader extends RecordReader<LongWritable,
       BytesWritable> {
39         private static final Log LOG = LogFactory.getLog(
              rawInputRecordReader.class);
40
41         private int recordLength;
42         private long start;
43         private long pos;
44         private long end;
45         private int globalSplitSize;
46         private long numRecordsRemainingInSplit;
47         private FSDataInputStream fileIn;
48         private Seekable filePosition;
49         private LongWritable key;
50         private BytesWritable value;
51         private boolean isCompressedInput;
52         private Decompressor decompressor;
53         private InputStream inputStream;
54
55         public rawInputRecordReader(int recordLength) {
56                 this.recordLength = recordLength;
57         }
58
59         @Override
60         public void initialize(InputSplit genericSplit,
              TaskAttemptContext context) throws IOException {
61                 FileSplit split = (FileSplit) genericSplit;
62                 Configuration job = context.getConfiguration();
63                 final Path file = split.getPath();
```

```
64                  if ( ((int) split.getLength()) > recordLength){
65                          globalSplitSize = recordLength;
66                  }
67                  else {
68                          globalSplitSize = (int) split.getLength();
69                  }
70                  initialize(job, split.getStart(), split.getLength(),
                        file);
71          }
72
73          public void initialize(Configuration job, long splitStart,
                long splitLength, Path file) throws IOException {
74                  start = splitStart;
75                  end = start + splitLength;
76                  long partialRecordLength = start % recordLength;
77                  long numBytesToSkip = 0;
78                  if (partialRecordLength != 0) {
79                          numBytesToSkip = globalSplitSize -
                                partialRecordLength;
80                  }
81
82                  final FileSystem fs = file.getFileSystem(job);
83                  fileIn = fs.open(file);
84
85                  CompressionCodec codec = new CompressionCodecFactory
                        (job).getCodec(file);
86                  if (null != codec) {
87                          isCompressedInput = true;
88                          decompressor = CodecPool.getDecompressor(
                                codec);
89                          CompressionInputStream cIn = codec.
                                createInputStream(fileIn, decompressor);
90                          filePosition = cIn;
91                          inputStream = cIn;
92                          numRecordsRemainingInSplit = Long.MAX_VALUE;
93                          LOG.info("Compressed input; cannot compute
                                number of records in the split");
94                  }
95                  else {
96                          fileIn.seek(start);
```

```
97                      filePosition = fileIn;
98                      inputStream = fileIn;
99                      System.out.println("end: " + end + " start:
                            " + start + " numBytesToSkip: " +
                            numBytesToSkip);
100                     long splitSize = end - start -
                            numBytesToSkip;
101                     //globalSplitSize = (int) (end - start -
                            numBytesToSkip);
102                     numRecordsRemainingInSplit = (splitSize +
                            recordLength -1)/recordLength;
103                     if (numRecordsRemainingInSplit < 0) {
104                             numRecordsRemainingInSplit =0;
105                     }
106                     LOG.info("Expecting " +
                            numRecordsRemainingInSplit + " records
                            each with a length of "
107                             + recordLength + " bytes in the
                                split with an effective size of "
                                 + splitSize
108                             + " bytes");
109
110             }
111             if (numBytesToSkip != 0) {
112                     start += inputStream.skip(numBytesToSkip);
113             }
114             this.pos = start;
115     }
116
117     @Override
118     public synchronized boolean nextKeyValue() throws
            IOException {
119             if (key == null) {
120                     key = new LongWritable();
121             }
122             if (value == null) {
123                     value = new BytesWritable(new byte[
                            globalSplitSize]);
124             }
125             boolean dataRead = false;
```

```
126                 value.setSize(globalSplitSize);
127                 byte[] record = value.getBytes();
128                 if (numRecordsRemainingInSplit > 0) {
129                         key.set(pos);
130                         int offset = 0;
131                         int numBytesToRead = globalSplitSize;
132                         int numBytesRead = 0;
133                         while (numBytesToRead > 0) {
134                                 numBytesRead = inputStream.read(
                                       record, offset, numBytesToRead);
135                                 if (numBytesRead == -1) {
136                                         break; //EOF
137                                 }
138                                 offset += numBytesRead;
139                                 numBytesToRead -= numBytesRead;
140                         }
141                         numBytesRead = globalSplitSize -
                                numBytesToRead;
142                         pos += numBytesRead;
143                         if (numBytesRead > 0) {
144                                 dataRead = true;
145                                 if (numBytesRead >= globalSplitSize)
                                         {
146                                         if (!isCompressedInput) {
147                                                 numRecordsRemainingInSplit
                                                        --;
148                                         }
149                                 }
150                         }
151                         else {
152                                 numRecordsRemainingInSplit = 0L;
153                         }
154                 }
155                 return dataRead;
156         }
157
158         @Override
159         public LongWritable getCurrentKey() {
160                 return key;
161         }
```

```
162
163         @Override
164         public BytesWritable getCurrentValue() {
165                 return value;
166         }
167
168         @Override
169         public synchronized float getProgress() throws IOException {
170                 if (start == end) {
171                         return 0.0f;
172                 }
173                 else {
174                         return Math.min(1.0f, (getFilePosition() -
                                start) / (float)(end - start));
175                 }
176         }
177
178         @Override
179         public synchronized void close() throws IOException {
180                 try {
181                         if (inputStream != null) {
182                                 inputStream.close();
183                                 inputStream = null;
184                         }
185                 }
186                 finally {
187                         if (decompressor != null) {
188                                 CodecPool.returnDecompressor(
                                        decompressor);
189                                 decompressor = null;
190                         }
191                 }
192         }
193
194         public long getPos() {
195                 return pos;
196         }
197
198         private long getFilePosition() throws IOException {
199                 long retVal;
```

```
200                 if (isCompressedInput && null != filePosition) {
201                         retVal = filePosition.getPos();
202                 }
203                 else {
204                         retVal = pos;
205                 }
206                 return retVal;
207         }
208 }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
WordCount Pseudo-code

This appendix contains pseudo-code for a MapReduce program. The most common basic MapReduce program is a word count program which is demonstrated in the below code. Full Source code for a MapReduce WordCount program may be found at the Apache tutorial webpage [80].

C.1 Word Count Pseudo-Code

```
1  map(String key, String value):
2          // key: document name
3          // value: input split contents
4          for each word in value:
5                  write(word, 1)
6
7  reduce(String intermediateKey, Iterator intermediateValues):
8          // key: a word
9          // values: list of counts
10         for each value in values:
11                 sum += value
12         write(key, sum)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
MapReduce ByteCount

This appendix contains complete code for the MapReduce byteCount program using both Int Array and HashMap.

D.1 Int Array ByteCount

Below is the code used to implement MapReduce byteCount program utilizing the Int Array.

```
1  package bytes;
2
3  import java.io.IOException;
4  import java.io.*;
5  import java.util.HashMap;
6  import java.util.Map;
7  import java.util.Iterator;
8  import java.util.Set;
9
10 import org.apache.hadoop.conf.Configuration;
11 import org.apache.hadoop.fs.Path;
12 import org.apache.hadoop.io.IntWritable;
13 import org.apache.hadoop.io.Text;
14 import org.apache.hadoop.io.BytesWritable;
15 import org.apache.hadoop.io.LongWritable;
16 import org.apache.hadoop.mapreduce.Job;
17 import org.apache.hadoop.mapreduce.Mapper;
18 import org.apache.hadoop.mapreduce.Reducer;
19 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
20 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
21
22 //      LOGGING
23 import org.apache.commons.logging.Log;
24 import org.apache.commons.logging.LogFactory;
25
26 //      TOOL
27 import org.apache.hadoop.util.Tool;
28 import org.apache.hadoop.util.ToolRunner;
```

```
29 import org.apache.hadoop.conf.Configured;
30
31 public class ByteCountIntArray extends Configured implements Tool {
32         private static final Log LOG = LogFactory.getLog(
               newByteCount.class);
33
34         public static class byteMapper extends Mapper<Object,
               BytesWritable, IntWritable, LongWritable> {
35               public void map(Object key, BytesWritable value,
                     Context context) throws IOException,
                     InterruptedException {
36
37                     byte[] byteArray = value.getBytes();
38
39                     int[] intArray = new int[256];
40
41                     for (int j=0; j< byteArray.length; j++){
42                             int byteValue = byteArray[j] & 0xFF;
43                             intArray[byteValue] += 1;
44                     }
45
46                     for(int i=0; i<=255; i++) {
47                             context.write(new IntWritable(i),
                                   new LongWritable(intArray[i]));
48                     }
49               }
50         }
51
52         public static class ByteSumReducer extends Reducer<
               IntWritable, LongWritable, IntWritable, LongWritable> {
53
54               private LongWritable result = new LongWritable();
55
56               public void reduce(IntWritable key, Iterable<
                     LongWritable> values, Context context) throws
                     IOException, InterruptedException {
57
58                     long sum = 0;
59                     for (LongWritable val : values) {
60
```

```
61                                  sum += val.get();
62                          }
63                          result.set(sum);
64                          context.write(key, result);
65                  }
66          }
67
68          public static void main(String[] args) throws Exception {
69                  int res = ToolRunner.run(new Configuration(), new
                        newByteCount(), args);
70                  System.exit(res);
71          }
72
73          @Override
74          public int run(String[] args) throws Exception {
75                  Configuration conf = this.getConf();
76                  conf.setInt(rawInputFormat.FIXED_RECORD_LENGTH,
                        536870912);
77                  Job job = new Job(conf, "byte count");
78                  job.setJarByClass(newByteCount.class);
79                  job.setInputFormatClass(rawInputFormat.class);
80                  job.setMapperClass(byteMapper.class);
81                  job.setCombinerClass(ByteSumReducer.class);
82                  job.setReducerClass(ByteSumReducer.class);
83                  job.setMapOutputKeyClass(IntWritable.class);
84                  job.setMapOutputValueClass(LongWritable.class);
85                  job.setOutputKeyClass(IntWritable.class);
86                  job.setOutputValueClass(LongWritable.class);
87                  FileInputFormat.addInputPath(job, new Path(args[0]))
                        ;
88                  FileOutputFormat.setOutputPath(job, new Path(args
                        [1]));
89                  return job.waitForCompletion(true) ? 0 : 1;
90
91          }
92  }
```

D.2 HashMap Byte Count

Below is the code used to implement MapReduce byteCount program utilizing a Java HashMap. Note that this approach is not recommended and we include it only for purposes of reproducing our memory analysis.

```
1  package bytes;
2
3  import java.io.IOException;
4  import java.io.*;
5  import java.util.HashMap;
6  import java.util.Map;
7  import java.util.Iterator;
8  import java.util.Set;
9
10 import org.apache.hadoop.conf.Configuration;
11 import org.apache.hadoop.fs.Path;
12 import org.apache.hadoop.io.IntWritable;
13 import org.apache.hadoop.io.Text;
14 import org.apache.hadoop.io.BytesWritable;
15 import org.apache.hadoop.io.LongWritable;
16 import org.apache.hadoop.mapreduce.Job;
17 import org.apache.hadoop.mapreduce.Mapper;
18 import org.apache.hadoop.mapreduce.Reducer;
19 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
20 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
21
22
23 //      LOGGING
24 import org.apache.commons.logging.Log;
25 import org.apache.commons.logging.LogFactory;
26
27 //      TOOL
28 import org.apache.hadoop.util.Tool;
29 import org.apache.hadoop.util.ToolRunner;
30 import org.apache.hadoop.conf.Configured;
31
32 public class ByteCountHashMap extends Configured implements Tool {
33         private static final Log LOG = LogFactory.getLog(
              newByteCount.class);
34
```

```
35          public static class byteMapper extends Mapper<Object,
                BytesWritable, IntWritable, LongWritable> {
36              private final static IntWritable one = new
                    IntWritable(1);
37              public void map(Object key, BytesWritable value,
                    Context context) throws IOException,
                    InterruptedException {
38
39                  HashMap<Integer, Integer> combinedMap = new
                        HashMap<Integer, Integer>();
40                  byte[] byteArray = value.getBytes();
41                  for (int j=0; j< byteArray.length; j++){
42                      int byteValue = byteArray[j] & 0xFF;
43                      if (combinedMap.containsKey(
                            byteValue)){
44                          int val = combinedMap.get(
                                byteValue);
45                          combinedMap.put(byteValue,
                                val+1);
46                      } else {
47                          combinedMap.put(byteValue,
                                1);
48                      }
49                  }
50                  for (Map.Entry<Integer, Integer> entry :
                        combinedMap.entrySet()){
51                      context.write(new IntWritable(entry.
                            getKey()), new LongWritable(entry
                            .getValue()));
52                  }
53              }
54          }
55
56          public static class ByteSumReducer extends Reducer<
                IntWritable, LongWritable, IntWritable, LongWritable> {
57
58              private LongWritable result = new LongWritable();
59
60              public void reduce(IntWritable key, Iterable<
                    LongWritable> values, Context context) throws
```

```
                    IOException, InterruptedException {
61
62                      long sum = 0;
63                      for (LongWritable val : values) {
64                              sum += val.get();
65                      }
66                      result.set(sum);
67                      context.write(key, result);
68              }
69      }
70
71      public static void main(String[] args) throws Exception {
72              int res = ToolRunner.run(new Configuration(), new
                    newByteCount(), args);
73              System.exit(res);
74      }
75
76      @Override
77      public int run(String[] args) throws Exception {
78              Configuration conf = this.getConf();
79              conf.setInt(rawInputFormat.FIXED_RECORD_LENGTH,
                    536870912);
80              Job job = new Job(conf, "byte count");
81              job.setJarByClass(newByteCount.class);
82              job.setInputFormatClass(rawInputFormat.class);
83              job.setMapperClass(byteMapper.class);
84              job.setCombinerClass(ByteSumReducer.class);
85              job.setReducerClass(ByteSumReducer.class);
86              job.setMapOutputKeyClass(IntWritable.class);
87              job.setMapOutputValueClass(LongWritable.class);
88              job.setOutputKeyClass(IntWritable.class);
89              job.setOutputValueClass(LongWritable.class);
90              FileInputFormat.addInputPath(job, new Path(args[0]))
                    ;
91              FileOutputFormat.setOutputPath(job, new Path(args
                    [1]));
92              return job.waitForCompletion(true) ? 0 : 1;
93
94      }
95  }
```

APPENDIX E:
MapReduce Bulk_Extractor Email Scanner

Below is MapReduce job code to execute be_scan email scanner on a Hadoop cluster using our rawInputFormat class as well `be_scan` and the Java interfaces to execute C++ bulk extractor libraries.

E.1 MapReduce Bulk Extractor Email

Usage of this code requires be_scan [76] to be installed.

```
1  package beScanner;
2
3  import java.io.IOException;
4  import java.io.*;
5  import java.util.HashMap;
6  import java.util.Map;
7  import java.util.Iterator;
8  import java.util.Set;
9  import java.util.Arrays;
10 import java.net.*;
11
12 import org.apache.hadoop.conf.Configuration;
13 import org.apache.hadoop.fs.Path;
14 import org.apache.hadoop.io.IntWritable;
15 import org.apache.hadoop.io.Text;
16 import org.apache.hadoop.io.BytesWritable;
17 import org.apache.hadoop.io.LongWritable;
18 import org.apache.hadoop.mapreduce.Job;
19 import org.apache.hadoop.mapreduce.Mapper;
20 import org.apache.hadoop.mapreduce.Reducer;
21 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
22 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
23 //      LOGGING
24 import org.apache.commons.logging.Log;
25 import org.apache.commons.logging.LogFactory;
26 //      TOOL
27 import org.apache.hadoop.util.Tool;
```

```
28 import org.apache.hadoop.util.ToolRunner;
29 import org.apache.hadoop.conf.Configured;
30
31 import org.apache.hadoop.filecache.DistributedCache;
32
33 public class beScan extends Configured implements Tool{
34
35         private static final Log LOG = LogFactory.getLog(beScan.
             class);
36
37         public static class beScanMapper extends Mapper<Object,
             BytesWritable, Text, IntWritable> {
38                 private final static IntWritable one = new
                    IntWritable(1);
39                 private Text emailAddress = new Text();
40
41                 public void setup (Context context) throws
                    IOException, InterruptedException {
42                         System.load((new File("libbe_scan_jni.so")).
                            getAbsolutePath());
43
44
45                 }
46
47                 public void map(Object key, BytesWritable value,
                    Context context) throws IOException,
                    InterruptedException {
48
49                         byte[] buffer1 = value.getBytes();
50
51                         edu.nps.deep.be_scan.BEScan scanner = new
                            edu.nps.deep.be_scan.BEScan("email",
                            buffer1, buffer1.length);
52                         edu.nps.deep.be_scan.Artifact artifact;
53
54                         artifact = scanner.next();
55                         while(!artifact.getArtifact().isEmpty()) {
56                                 emailAddress.set(artifact.
                                    getArtifact());
57                                 context.write(emailAddress, one);
```

```
58                              artifact = scanner.next();
59                      }
60
61              }
62      }
63
64      public static class beScanReducer extends Reducer<Text,
            IntWritable, Text, IntWritable> {
65              private IntWritable result = new IntWritable();
66
67              public void reduce(Text key, Iterable<IntWritable>
                    values, Context context) throws IOException,
                    InterruptedException {
68                      int sum = 0;
69                      for (IntWritable val : values) {
70                              sum += val.get();
71                      }
72                      result.set(sum);
73                      context.write(key, result);
74              }
75
76
77
78      }
79
80      public static void main(String[] args) throws Exception {
81              int res = ToolRunner.run(new Configuration(), new
                    beScan(), args);
82              System.exit(res);
83      }
84
85      @Override
86      public int run(String[] args) throws Exception {
87              Configuration conf = this.getConf();
88              conf.setInt(rawInputFormat.FIXED_RECORD_LENGTH,
                    536870912); //512MB recordlength in bytes
89              Job job = new Job(conf, "BE Scanner");
90              job.setJarByClass(beScan.class);
91              job.setInputFormatClass(rawInputFormat.class);
92              job.setMapperClass(beScanMapper.class);
```

```
93                  job.setCombinerClass(beScanReducer.class);
94                  job.setReducerClass(beScanReducer.class);
95                  job.setMapOutputKeyClass(Text.class);
96                  job.setMapOutputValueClass(IntWritable.class);
97                  job.setOutputKeyClass(Text.class);
98                  job.setOutputValueClass(IntWritable.class);
99                  job.addCacheFile(new URI("hdfs://hadmin-1-33a.ib.
                        grace.cluster:8020/user/tjandrze/libraries/
                        libbe_scan_jni.so.0.0.0#libbe_scan_jni.so"));
100                 FileInputFormat.addInputPath(job, new Path(args[0]))
                        ;
101                 FileOutputFormat.setOutputPath(job, new Path(args
                        [1]));
102                 return job.waitForCompletion(true) ? 0 : 1;
103
104         }
105 }
```

APPENDIX F:
MapReduce ByteCount Results

F.1 MapReduce ByteCount Result Table

Tables F.1 and F.2 represent the byte frequency of RDC sorted byte value.

Table F.1. ByteCount Bytes 0-171

Byte	Byte Frequency	Byte	Byte Frequency	Byte	Byte Frequency	Byte	Byte Frequency
0	63273879033072	43	208262015370	86	238808219211	129	228893989146
1	734295502830	44	226331516762	87	230021555544	130	204847502458
2	492826566372	45	262526671948	88	221676764015	131	270808291723
3	396546091375	46	244307775869	89	200052280236	132	224393496505
4	418845885918	47	242232684273	90	343891098770	133	244748668795
5	304600414267	48	570435823506	91	189972894685	134	193602304291
6	311225415223	49	313376899440	92	213905952188	135	181532065531
7	297314486824	50	284049867514	93	205651869281	136	215686178542
8	383449628710	51	316623710149	94	195784862524	137	278700923711
9	269047851725	52	268118049361	95	228363625902	138	189534177745
10	314633894433	53	271161518976	96	241785533639	139	437057702605
11	238842543530	54	246400508982	97	347070445994	140	198706678259
12	308563927742	55	240258653642	98	228608146027	141	264790855826
13	266646197630	56	267028280773	99	274455994215	142	182229736158
14	237917722531	57	258790573407	100	275612396625	143	184412321291
15	323117437993	58	254980940211	101	417524034178	144	261594198865
16	360945179962	59	226880224354	102	257105817870	145	185318395286
17	261242861971	60	231737096390	103	227034040195	146	190125549635
18	234532691466	61	250906563340	104	260822368387	147	179250282358
19	215301111128	62	231281848497	105	324512093963	148	229802166020
20	259932073057	63	269076544603	106	251447347901	149	186327318058
21	230464240083	64	322847361867	107	208445958642	150	177424112083
22	207649976670	65	381184286916	108	279799172447	151	170208157286
23	203098395553	66	285567000946	109	241032252997	152	185414780978
24	258054441375	67	255479826947	110	304882106113	153	182176365894
25	205818201769	68	299021992591	111	325175921573	154	180400694730
26	200500276838	69	331932759619	112	360203249171	155	169175419877
27	195976403880	70	284496812692	113	206924937414	156	189023810848
28	221392688465	71	212823478273	114	323885123209	157	176640016113
29	194063668849	72	249992756974	115	311799164908	158	184337867667
30	203476380924	73	238283995524	116	405259365557	159	180232454755
31	242456459054	74	206024019414	117	310507668844	160	225115980730
32	716338085559	75	193535564813	118	220185426010	161	190363757989
33	246711736634	76	236437080955	119	222407823807	162	178868639615
34	272952917264	77	255381638768	120	228475496218	163	185549236917
35	233399290430	78	222511180999	121	218501476201	164	187438626304
36	300116349930	79	218734828494	122	192490271846	165	189423408651
37	205697393972	80	333183376938	123	193647647178	166	172556552337
38	204778101364	81	225563655700	124	209355960882	167	178992659289
39	202296568445	82	242013408285	125	213903634895	168	196724030281
40	237301150880	83	274087125110	126	267102555096	169	180324147161
41	206961539210	84	245774913872	127	313306384399	170	452140606558
42	205008623311	85	292283580950	128	382474487276	171	181537150838

Table F.2. ByteCount Bytes 171-255

Byte	Byte Frequency	Byte	Byte Frequency
172	183661904540	214	182086893190
173	228051795035	215	177173773497
174	171957407832	216	191614166918
175	186286053543	217	174139708899
176	204189250266	218	212875371499
177	178553934662	219	185191974683
178	171937232247	220	188479780145
179	167760651990	221	183443165147
180	178876375387	222	185375289755
181	184174144724	223	212584923600
182	180061894867	224	263962689583
183	178037748743	225	228236840319
184	199695598008	226	193868862363
185	185354406872	227	186513348817
186	176627881743	228	190268468446
187	209864245870	229	183085039581
188	182104933905	230	180708548368
189	193525142486	231	196235299639
190	190401123812	232	315853574905
191	209960099080	233	219261485825
192	317879908575	234	190758707589
193	216022687611	235	210061514770
194	205709445265	236	208605731649
195	215753105461	237	184839959887
196	207957114211	238	188117519609
197	177536990396	239	210693355629
198	195581530751	240	261548387121
199	208917252438	241	209865464453
200	260518703062	242	201474835858
201	199832353182	243	199179267256
202	239798425278	244	203441996386
203	176828509880	245	192772838665
204	258770747154	246	1351928628522
205	181881426338	247	217676577968
206	185358064624	248	239968603243
207	185940839466	249	242029718491
208	209475741946	250	204641541837
209	182396071346	251	225354199703
210	184046843323	252	257422593513
211	180889000885	253	245280015393
212	181009463175	254	292508008283
213	178366056202	255	3183095937209

F.2 MapReduce ByteCount Frequency Sorted Table

Tables F.3 and F.4 below represent the byte frequency of RDC sorted byte frequency.

Table F.3. ByteCount 150 Least Frequent Bytes

Byte	Byte Frequency	Byte	Byte Frequency	Byte	Byte Frequency
179	167760651990	219	185191974683	242	201474835858
155	169175419877	145	185318395286	39	202296568445
151	170208157286	185	185354406872	23	203098395553
178	171937232247	206	185358064624	244	203441996386
174	171957407832	222	185375289755	30	203476380924
166	172556552337	152	185414780978	176	204189250266
217	174139708899	163	185549236917	250	204641541837
186	176627881743	207	185940839466	38	204778101364
157	176640016113	175	186286053543	130	204847502458
203	176828509880	149	186327318058	42	205008623311
215	177173773497	227	186513348817	93	205651869281
150	177424112083	164	187438626304	37	205697393972
197	177536990396	238	188117519609	194	205709445265
183	178037748743	220	188479780145	25	205818201769
213	178366056202	156	189023810848	74	206024019414
177	178553934662	165	189423408651	113	206924937414
162	178868639615	138	189534177745	41	206961539210
180	178876375387	91	189972894685	22	207649976670
167	178992659289	146	190125549635	196	207957114211
147	179250282358	228	190268468446	43	208262015370
182	180061894867	161	190363757989	107	208445958642
159	180232454755	190	190401123812	236	208605731649
169	180324147161	234	190758707589	199	208917252438
154	180400694730	216	191614166918	124	209355960882
230	180708548368	122	192490271846	208	209475741946
211	180889000885	245	192772838665	187	209864245870
212	181009463175	189	193525142486	241	209865464453
135	181532065531	75	193535564813	191	209960099080
171	181537150838	134	193602304291	235	210061514770
205	181881426338	123	193647647178	239	210693355629
214	182086893190	226	193868862363	223	212584923600
188	182104933905	29	194063668849	71	212823478273
153	182176365894	198	195581530751	218	212875371499
142	182229736158	94	195784862524	125	213903634895
209	182396071346	27	195976403880	92	213905952188
229	183085039581	231	196235299639	19	215301111128
221	183443165147	168	196724030281	136	215686178542
172	183661904540	140	198706678259	195	215753105461
210	184046843323	243	199179267256	193	216022687611
181	184174144724	184	199695598008	247	217676577968
158	184337867667	201	199832353182	121	218501476201
143	184412321291	89	200052280236	79	218734828494
237	184839959887	26	200500276838	233	219261485825

Byte	Byte Frequency
118	220185426010
28	221392688465
88	221676764015
119	222407823807
78	222511180999
132	224393496505
160	225115980730
251	225354199703
81	225563655700
44	226331516762
59	226880224354
103	227034040195
173	228051795035
225	228236840319
95	228363625902
120	228475496218
98	228608146027
129	228893989146
148	229802166020
87	230021555544
21	230464240083

Table F.4. ByteCount 106 Most Frequent Bytes

Byte	Byte Frequency	Byte	Byte Frequency
62	231281848497	63	269076544603
60	231737096390	131	270808291723
35	233399290430	53	271161518976
18	234532691466	34	272952917264
76	236437080955	83	274087125110
40	237301150880	99	274455994215
14	237917722531	100	275612396625
73	238283995524	137	278700923711
86	238808219211	108	279799172447
11	238842543530	50	284049867514
202	239798425278	70	284496812692
248	239968603243	66	285567000946
55	240258653642	85	292283580950
109	241032252997	254	292508008283
96	241785533639	7	297314486824
82	242013408285	68	299021992591
249	242029718491	36	300116349930
47	242232684273	5	304600414267
31	242456459054	110	304882106113
46	244307775869	12	308563927742
133	244748668795	117	310507668844
253	245280015393	6	311225415223
84	245774913872	115	311799164908
54	246400508982	127	313306384399
33	246711736634	49	313376899440
72	249992756974	10	314633894433
61	250906563340	232	315853574905
106	251447347901	51	316623710149
58	254980940211	192	317879908575
77	255381638768	64	322847361867
67	255479826947	15	323117437993
102	257105817870	114	323885123209
252	257422593513	105	324512093963
24	258054441375	111	325175921573
204	258770747154	69	331932759619
57	258790573407	80	333183376938
20	259932073057	90	343891098770
200	260518703062	97	347070445994
104	260822368387	112	360203249171
17	261242861971	16	360945179962
240	261548387121	65	381184286916
144	261594198865	128	382474487276
45	262526671948	8	383449628710
224	263962689583	3	396546091375
141	264790855826	116	405259365557
13	266646197630	101	417524034178
56	267028280773	4	418845885918
126	267102555096	139	437057702605
52	268118049361	170	452140606558
9	269047851725	2	492826566372

Byte	Byte Frequency
48	570435823506
32	716338085559
1	734295502830
246	1351928628522
255	3183095937209
0	63273879033072

APPENDIX G:
Calculate TF-IDF Python Program

Below is a sample python program to calculate TF-IDF values for an email address and hard drive image in the RDC. This program relies on the results from the Mapreduce bulk extractor program in a CSV file.

G.1 Calculate TF-IDF

```
#!/usr/bin/python
import sys
import csv
import math

email = ":" + sys.argv[1]
fileName = sys.argv[2]

emailDict = {}
fileDict = {}
drivesInCorpus = 3088.0

with open('/path/to/mapreduce/bulk/extractor/results.csv', mode='r')
    as infile:
        reader = csv.reader(infile)
        for row in reader:
                if email in row[0]:
                        emailDict[row[0]] = row[1]
                if fileName in row[0]:
                        fileDict[row[0]] = row[1]

numDrivesEmailFoundIn = float(len(emailDict))
emailFrequency = float(emailDict[fileName + email])
totalEmailsInDrive = 0.0
for key, value in fileDict.iteritems():
        totalEmailsInDrive += float(value)
inverseDocFreq = math.log10(drivesInCorpus/numDrivesEmailFoundIn)
termFreq = emailFrequency/totalEmailsInDrive
```

```
28 tf_idf = termFreq * inverseDocFreq
29 print(sys.argv[1] +" in " + fileName + " has a TF-IDF value of: ")
30 print(tf_idf)
```

APPENDIX H:
Writing Bulk_Extractor MapReduce

The general flow of this MapReduce job is as follows. The Mapper portion will accept as input key-value pairs, where value is the contents of a disk image inputSplit from a file read from HDFS. The Mapper then loads the be_scan library which is used to extract email addresses from the bytes stored as the value. Any email address found is sent to the Reducer which counts each unique email address, similar to the WordCount program counting words in Appendix C.

The record size of value is customizable using the `FIXED_RECORD_LENGTH` parameter. This program sets the parameter to the same size as the data blocksize, 512MiB, in the job configuration portion. This value was chosen because initial analysis demonstrated better performance when record size and data blocksize were the same, though this analysis was not exhaustive.

The Mapper function will load `libbe_scan_jni.so` from a user specified HDFS path. This program loads `libbe_scan_jni.so` from `/user/tjandrze/libraries/` which is populated with `libbe_scan_jni.so.0.0.0` from the be_scan build directory using the `hdfs dfs -put` command. To make this library available in the path of the MapReduce job, Hadoop DistributedCache [81] is required. This will distribute the library from the HDFS path to each datanode at runtime to the path of the running Mappers, which then load the library. The distribution and loading of the library is done via the commands listed below. Line 1 utilizes distributedCache to distribute the library while line 4 loads this library into the Mapper.

```
1 job.addCacheFile(new URI("hdfs://hadmin-1-33a.ib.grace.cluster:8020/
    user/tjandrze/libraries/libbe_scan_jni.so.0.0.0#libbe_scan_jni.so
    "));
2
3 public void setup (Context context) throws IOException,
    InterruptedException {
4         System.load((new File("libbe_scan_jni.so")).getAbsolutePath
              ());
5 }
```

With the library loaded into the MapReduce job, the map function is able to successfully use the BEScan and Artifact classes found in the edu directory from Figure 4.2. The map function code below calls the BEScan class and the Artifact class, which extract email addresses via the distributed be_scan library. Line three stores the bytes of value into a byte array which is sent as an argument during scanner object creation in line five. Line six defines a new Artifact object where an artifact is what is returned from the scanner(i.e., an email address for the email scanner). Line eight uses the `next()` method from the BEScan class which directs the program to search for the next artifact (email address). In this case it is to find the first artifact, if one exists in value. Lines nine through 12 loop through each non-empty artifact and set the `Text()` object and emailAddress to the contents of the artifact. This is required because the Mapper can only write or send to the Reducer Hadoop writable types, and Artifact is not a Hadoop writable type. Similar to the wordCount program, line 11 writes the email address and a "one," which is sent to the Reducer. The loop continues by directing the scanner to locate the next artifact. This continues until all bytes in the value are read.

```
1  public void map(Object key, BytesWritable value, Context context)
       throws IOException, InterruptedException {
2
3          byte[] buffer1 = value.getBytes();
4
5          edu.nps.deep.be_scan.BEScan scanner = new edu.nps.deep.
               be_scan.BEScan("email", buffer1, buffer1.length);
6          edu.nps.deep.be_scan.Artifact artifact;
7
8          artifact = scanner.next();
9          while(!artifact.getArtifact().isEmpty()) {
10                 emailAddress.set(artifact.getArtifact());
11                 context.write(emailAddress, one);
12                 artifact = scanner.next();
13         }
14 }
```

The Reducer function receives as key-value pair the `Text()` emailAddress object as the key and an `IntWritable()` object set to the number one. These keys and values are received from the Mapper functions, which write them to intermediate local files. They are then

reduced or summed on all identical email addresses as seen in lines three through five. The Reducer then writes the unique email address and count to the output directory specified by the user.

```
1 public void reduce(Text key, Iterable<IntWritable> values, Context
      context) throws IOException, InterruptedException {
2         int sum = 0;
3         for (IntWritable val : values) {
4                 sum += val.get();
5         }
6         result.set(sum);
7         context.write(key, result);
8 }
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] V. Roussev and G. G. Richard III, "Breaking the performance wall: The case for distributed digital forensics," in *Proceedings of the 2004 Digital Forensics Research Workshop*, 2004, vol. 94.

[2] S. L. Garfinkel, "Digital forensics research: The next 10 years," *Digital Investigation*, vol. 7, pp. S64–S73, 2010.

[3] D. Edwards, "Tech refresh for the forensic analysis toolkit," *SANS Institute InfoSec Reading Room*, 2010.

[4] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct sector hashes for target file detection," *Computer*, vol. 45, no. 12, pp. 28–35, 2012.

[5] M. Pollitt, *A History of Digital Forensics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–15. Available: http://dx.doi.org/10.1007/978-3-642-15506-2_1

[6] C. Stoll, *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*. New York, New York: Simon and Schuster, 2005.

[7] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *Science*, vol. 332, no. 6025, pp. 60–65, 2011.

[8] G. Palmer, "A road map for digital forensic research," in *Proceedings of Digital Forensic Research Conference*, Utica, New York, 2001, pp. 14–18.

[9] Digital Forensics Association. Formal education: College education in digital forensics. [Online]. Available: http://www.digitalforensicsassociation.org/formal-education/. Accessed April 1, 2017.

[10] S. L. Garfinkel, "Digital media triage with bulk data analysis and bulk_extractor," *Computers & Security*, vol. 32, pp. 56–72, 2013.

[11] S. L. Garfinkel, "Forensic feature extraction and cross-drive analysis," *Digital Investigation*, vol. 3, pp. 71–81, 2006.

[12] ForensicsWiki. Forensics file formats. [Online]. Available: http://www.forensicswiki.org/wiki/Category:Forensics_File_Formats. Accessed July 1, 2017.

[13] Libewf: Library to access the Expert Witness Compression Format(EWF). libewf. [Online]. Available: https://github.com/libyal/libewf. Accessed July 1, 2017.

[14] Digital Corpora. Real Data Corpus. [Online]. Available: http://digitalcorpora.org/. Accessed April 21, 2017.

[15] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986.

[16] TFIDF. What does tf-idf mean? [Online]. Available: http://www.tfidf.com. Accessed August 27, 2017.

[17] S. Madden, "From databases to big data," *IEEE Internet Computing*, vol. 16, no. 3, pp. 4–6, 2012.

[18] D. Laney, "3d data management: Controlling data volume, velocity and variety," *META Group Research Note*, vol. 6, p. 70, 2001.

[19] J. S. Ward and A. Barker, "Undefined by data: A survey of big data definitions," *ArXiv Preprint ArXiv:1309.5821*, 2013.

[20] M. A. Beyer and D. Laney, "The importance of big data: A definition," *Stamford, CT: Gartner*, pp. 2014–2018, 2012.

[21] NIST. NIST Big Data Public Working Group (NBD-PWG). [Online]. Available: https://bigdatawg.nist.gov/home.php. Accessed April 14, 2017.

[22] O'luanaigh, Cian. CERN Data Centre passes 100 petabytes. [Online]. Available: https://home.cern/about/updates/2013/02/cern-data-centre-passes-100-petabytes. Accessed April 16, 2017.

[23] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money, "Big data: Issues and challenges moving forward," in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE, 2013, pp. 995–1004.

[24] Guidance Software. EnCase Forensic. [Online]. Available: https://www.guidancesoftware.com/encase-forensic. Accessed April 16, 2017.

[25] Access Data. Forensic Toolkit(FTK). [Online]. Available: http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk. Accessed April 16, 2017.

[26] Carrier, Brian. The Sleuth Kit(TSK). [Online]. Available: https://www.sleuthkit.org/. Accessed April 16, 2017.

[27] G. G. Richard III and V. Roussev, "Digital forensics tools: the next generation," *Digital Crime and Forensic Science in Cyberspace*, pp. 76–91, 2006.

[28] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *AcM sIGMoD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[29] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[30] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 2014.

[31] Apache. Hadoop. [Online]. Available: http://hadoop.apache.org. Accessed April 21, 2017.

[32] T. White, *Hadoop: The Definitive Guide 4th Edition [M]*. Sebastopol, California: O'Reilly Media, PP, 2009.

[33] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, no. 5. ACM, 2003, vol. 37, pp. 29–43.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[35] A. Luckow, I. Paraskevakos, G. Chantzialexiou, and S. Jha, "Hadoop on hpc: Integrating hadoop and pilot-based dynamic resource management," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1607–1616.

[36] Kranz, Chris. Data Protection: RAID, erasure coding or replication. [Online]. Available: http://www.hedviginc.com/blog/data-protection-raid-erasure-coding-or-replication. Accessed April 28, 2017.

[37] Eadline, Douglas. Is hadoop the new HPC? [Online]. Available: http://www.admin-magazine.com/HPC/Articles/Is-Hadoop-the-New-HPC. Accessed April 28, 2017.

[38] D. Quick and K.-K. R. Choo, "Impacts of increasing volume of digital forensic data: A survey and future research challenges," *Digital Investigation*, vol. 11, no. 4, pp. 273–294, 2014.

[39] K. Nance, B. Hay, and M. Bishop, "Digital forensics: Defining a research agenda," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. IEEE, 2009, pp. 1–6.

[40] S. Raghavan, "Digital forensic research: Current state of the art," *CSI Transactions on ICT*, vol. 1, no. 1, pp. 91–114, 2013.

[41] N. M. Karie and H. S. Venter, "Taxonomy of challenges for digital forensics," *Journal of Forensic Sciences*, vol. 60, no. 4, pp. 885–893, 2015.

[42] V. R. Ambhire and B. Meshram, "Digital forensic tools," *IOSR Journal of Engineering*, vol. 2, no. 3, pp. 392–398, 2012.

[43] D. Manson, A. Carlin, S. Ramos, A. Gyger, M. Kaufman, and J. Treichelt, "Is the open way a better way? digital forensics using open source tools," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 2007, pp. 266b–266b.

[44] Johns, Justin. Processing Manager. [Online]. Available: https://support.accessdata.com/hc/en-us/articles/206870837-Processing-Manager. Accessed May 29, 2017.

[45] Bone, Brendan. Configuring distributed processing in FTK/AD Enterprise/AD Lab. [Online]. Available: https://support.accessdata.com/hc/en-us/articles/211517937-Configuring-Distributed-Processing-in-FTK-AD-Enterprise-AD-Lab. Accessed May 29, 2017.

[46] Lee, Erika. AccessData delivers on distributed processing capabilities with its computer forensics technology. [Online]. Available: https://ad-pdf.s3.amazonaws.com/FTK_3.0.4_Distributed_Processing.pdf. Accessed May 29, 2017.

[47] D. Lillis, B. Becker, T. O'Sullivan, and M. Scanlon, "Current challenges and future research areas for digital forensic investigation," *arXiv preprint arXiv:1604.03850*, 2016.

[48] V. Roussev, C. Quates, and R. Martell, "Real-time digital forensics and triage," *Digital Investigation*, vol. 10, no. 2, pp. 158–167, 2013.

[49] V. Roussev, "Building open and scalable digital forensic tools," in *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*. IEEE, 2011, pp. 1–6.

[50] "Divide & conquer: Overcoming computer forensic backlog through distributed processing and division of labor," White Paper, AccessData Corporation, 2010.

[51] D. Ayers, "A second generation computer forensic analysis system," *Digital Investigation*, vol. 6, pp. S34–S42, 2009.

[52] V. Roussev, L. Wang, G. G. R. Iii, and L. Marziale, "Mmr: A platform for large-scale forensic computing," 2009.

[53] Carrier, Brian. The Sleuth Kit(TSK) hadoop framework. [Online]. Available: http://www.sleuthkit.org/tsk_hadoop/. Accessed May 30, 2017.

[54] C. Miller, D. Glendowne, D. Dampier, and K. Blaylock, "Forensicloud: An architecture for digital forensic analysis in the cloud," *Journal of Cyber Security*, vol. 3, pp. 231–262, 2014.

[55] N. Gunther, P. Puglia, and K. Tomasette, "Hadoop superlinear scalability," *Queue*, vol. 13, no. 5, p. 20, 2015.

[56] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs. scale-out for hadoop: Time to rethink?" in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 20.

[57] Z. Li and H. Shen, "Performance measurement on scale-up and scale-out hadoop with remote and local file systems," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 456–463.

[58] L. Marziale, G. G. Richard, and V. Roussev, "Massive threading: Using gpus to increase the performance of digital forensics tools," *Digital Investigation*, vol. 4, pp. 73–81, 2007.

[59] Carrier, Brian. scapel. [Online]. Available: https://github.com/sleuthkit/scalpel. Accessed July 16, 2017.

[60] S. Collange, Y. S. Dandass, M. Daumas, and D. Defour, "Using graphics processors for parallelizing hash-based data carving," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. IEEE, 2009, pp. 1–10.

[61] J. Lee, S. Un, and D. Hong, "High-speed search using tarari content processor in digital forensics," *Digital Investigation*, vol. 5, pp. S91–S95, 2008.

[62] J. Bengtsson, "Parallel password cracker: A feasibility study of using linux clustering technique in computer forensics," in *Digital Forensics and Incident Analysis, 2007. WDFIA 2007. Second International Workshop on*. IEEE, 2007, pp. 75–82.

[63] S. Alharbi, B. Moa, J. Weber-Jahnke, and I. Traore, "High performance proactive digital forensics," in *Journal of Physics: Conference Series*, no. 1. IOP Publishing, 2012, vol. 385, p. 012003.

[64] W. Alink, "Xiraf: An xml information retrieval approach to digital forensics," 2005.

[65] W. Alink, R. Bhoedjang, P. A. Boncz, and A. P. de Vries, "Xiraf–xml-based indexing and querying for digital forensics," *Digital Investigation*, vol. 3, pp. 50–58, 2006.

[66] R. A. Bhoedjang, A. R. van Ballegooij, H. M. van Beek, J. C. van Schie, F. W. Dillema, R. B. van Baar, F. A. Ouwendijk, and M. Streppel, "Engineering an online computer forensic service," *Digital Investigation*, vol. 9, no. 2, pp. 96–108, 2012.

[67] H. van Beek, E. van Eijk, R. van Baar, M. Ugen, J. Bodde, and A. Siemelink, "Digital forensics as a service: Game on," *Digital Investigation*, vol. 15, pp. 20–38, 2015.

[68] H. van Beek, "Digital forensics as a service: An update," in *Proceedings of Digital Forensic Research Workshop*, Seattle, Washington, 2016.

[69] J. Lee and S. Un, "Digital forensics as a service: A case study of forensic indexed search," in *ICT Convergence (ICTC), 2012 International Conference on*. IEEE, 2012, pp. 499–503.

[70] Altheide, Cory and Berggren, Johan. Turbinia: Cloud-scale forensics. [Online]. Available: https://github.com/google/turbinia. Accessed May 31, 2017.

[71] Apache Software Foundation. Hadoop MapReduce next generation - Cluster Setup. [Online]. Available: https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-common/ClusterSetup.html. Accessed July 5, 2017.

[72] ScehdMD. SLURM workload manager. [Online]. Available: https://slurm.schedmd.com/. Accessed August 04, 2017.

[73] Apache Hadoop. LineRecordReader.java. [Online]. Available: https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/lib/input/LineRecordReader.java. Accessed July 23, 2017.

[74] Apache Hadoop. FixedLengthInputFormat.java. [Online]. Available: https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/lib/input/FixedLengthInputFormat.java. Accessed July 23, 2017.

[75] Apache Hadoop. InputFormat. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/api/org/apache/hadoop/mapred/InputFormat.html. Accessed July 24, 2017.

[76] Allen, Bruce. be_scan. [Online]. Available: https://github.com/NPS-DEEP/be_scan. Accessed July 1, 2017.

[77] Apache Software Foundation. Partitioning your job into maps and reduces. [Online]. Available: https://wiki.apache.org/hadoop/HowManyMapsAndReduces. Accessed July 5, 2017.

[78] Oracle. Class HashMap. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html. Accessed August 19, 2017.

[79] Janglo. Our Story. [Online]. Available: http://www.janglo.net/content/view/136706/9999/. Accessed August 25, 2017.

[80] Apache. MapReduce Tutorial. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. Accessed April 21, 2017.

[81] Apache Hadoop. DistributedCache. [Online]. Available: https://hadoop.apache.org/docs/r2.6.3/api/org/apache/hadoop/filecache/DistributedCache.html. Accessed July 25, 2017.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California