University of Rhode Island
DigitalCommons@URI

Open Access Dissertations

2019

SPARTA: SYSTEM FOR PORTABLE ACQUISTION WITH REAL-TIME ANALYSIS

Joseph Greenfield
University of Rhode Island, joseph.greenfield@gmail.com

Follow this and additional works at: https://digitalcommons.uri.edu/oa_diss

Terms of Use
All rights reserved under copyright.

Recommended Citation
Greenfield, Joseph, "SPARTA: SYSTEM FOR PORTABLE ACQUISTION WITH REAL-TIME ANALYSIS" (2019). *Open Access Dissertations.* Paper 1082.
https://digitalcommons.uri.edu/oa_diss/1082

This Dissertation is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu. For permission to reuse copyrighted content, contact the author directly.

SPARTA: SYSTEM FOR PORTABLE ACQUISTION WITH REAL-TIME ANALYSIS

BY

JOSEPH GREENFIELD

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2019

DOCTOR OF PHILOSOPHY DISSERTATION

OF

JOSEPH GREENFIELD

APPROVED:

Dissertation Committee:

Major Professor Victor Fay-Wolfe

Lisa DiPippo

Yan Sun

Nasser H. Zawia
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2019

ABSTRACT

The aim of this thesis is to design, develop and test a new portable system for digital forensics imaging with real-time analysis over every live file. Currently large magnetic hard drives are infeasible to perform sequential imaging taking over 40 hours to complete before beginning with any forensic analysis. Attempted approaches included performing a limited (sparse) collection and performing a distributed live analysis using a high-end server environment, neither of which would be sufficient for field use. I designed and developed the code to test the system and developed comprehensive testing scenarios. I show that magnetic disk fragmentation has a direct, mostly linear impact over the speed at which a disk can be imaged and every live file be processed simultaneously. I show that RAM has a near exponential impact on simultaneous magnetic disk forensic imaging with all live file processing. I demonstrate that CASE/UCO has the potential to be the interoperable file format for digital forensics metadata exchange. I also demonstrate that a system for simultaneous forensic disk imaging with all live file analysis can be assembled with commercial off-the-shelf parts for less than $1000.

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor, Professor Victor Fay-Wolfe. Not many advisors would take on a part-time Ph. D. student, let alone one all the way across the country. His advice, guidance and support were invaluable while engaging in the practical research elements and writing of this thesis. Given my unique circumstance, his patience was necessary for this thesis to come to fruition.

Aside from my advisor, I would like to thank the rest of my thesis committee: Professor Lisa DiPippo, Professor Yan Sun and Professor Qing Yang for their comments, feedback and encouragement throughout the process.

I would like to thank the many members of the digital forensics research community, especially Dr. Bradley Schatz for providing ideas and insight into digital forensics research that helped formulate the problems tackled in this thesis.

I would also like to thank my professional mentor and business partner, Brad Maryman. I will always be grateful that he took me under his wing when I first engaged professionally in digital forensics, and I will always rely upon his guidance and advice.

Last, but absolutely not least, I would like to thank my family: my mother Sally for providing unconditional support, my father Jon for always showing me the value of investing in myself through education, my daughters Valentina and Ariella for always making me smile and laugh.

But most of all, this work is dedicated to my wife Fernanda. Without her by my side, nothing in my life, professional or personal, would be possible.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Digital forensic relies upon the scientific examination of digital evidence for use in court proceedings. The basic standard that is used for preservation of digital evidence is a bitstream image, which is a bit-for-bit copy of a source medium called a forensic image. When traditional hard drives were small in capacity, making a forensic image onsite was relatively straightforward and not time intensive. However, in the last decade, storage sizes have grown to the point that even making a basic forensic image is can take days to complete, which substantially delays investigations.

In 2013, Eric Zimmerman performed comprehensive testing of imaging speeds for a 1 terabyte hard drive. The fastest speeds achieved was a rate of approximately 6.6 Gigabytes per minute (GB/min) for imaging. [1] A commercially available 16 Terabyte magnetic hard drive would take a little over 40 hours to image before any data processing could be performed, since currently no forensic imaging tool performs simultaneous analysis of data while creating the forensic bitstream image.

In this dissertation I describe how I created a digital forensic imaging process that addresses the excessive delays that the imaging process can cause in investigations by introducing simultaneous live allocated file processing capabilities based on hardware specification and source drive fragmentation. I first created a non-distributed forensic imaging device (hardware and software), called SPARTA, that creates a CASE/UCO metadata file containing the hash analysis and signature analysis of every live allocated file in parallel with the bitstream image file creation. This system of

parallel processing has imaging speeds equal to leading commercial and open-source tools. Once implemented, I tested SPARTA against a standard lab-built dataset that is fragmented to precise percentages. In doing so, I produced interesting results showing how the system bottleneck will be I/O bus speed and disk fragmentation, and how the disk fragmentation affects the amount of RAM and processing cores necessary, page faults against the swap space, and the overall speed of output. From this I derive results as to what is the relationship between disk fragmentation, RAM and processing cores such that disk imaging performance is not severely impacted by core forensic processing in parallel.

The overall goals for this research were:

1. Correctness – To develop a portable digital forensic imaging system that will create a correct bitstream forensic image file with simultaneous correct processing of each logical file.

2. Efficiency – To develop a portable digital forensic imaging system that will perform a bitstream copy of a source medium with simultaneous live file processing

3. Cost – To develop a system with commercial-off-the-shelf (COTS) parts with reasonable cost (less than $1000).

4. Cross-Platform Compatibility – To develop a system with a metadata file output that can be easily imported into industry standard tools.

Chapter 2 provides background on this research including work that addresses related problems in digital forensics imaging, as well as foundational work for my project. Chapter 3 describes the development of the SPARTA prototype. Chapter 4 describes the experimental design and tests that I performed. Chapter 4 will also the

articulate the analysis of the results and how I met the four goals of the work listed above. Chapter 5 concludes by summarizing the contributions of this project and future work for extending the research.

CHAPTER 2

BACKGROUND AND REVIEW OF LITERATURE

2.1 Digital Forensics Imaging

The National Institute of Standards and Technology (NIST) has one definition of digital forensics as "the application of science to the identification, collection, examination and analysis, of data while preserving the integrity of the information and maintaining a strict chain of custody for the data." [2] Forensics is especially necessary when the examined evidence is to be used in court proceedings, where preservation of evidence is critical to establish authenticity and provenance for the examined artifacts. The basic standard that is used for preservation of digital evidence is a bitstream image, which is a bit-for-bit copy of a source medium called a *forensic image*. [2]

When traditional hard drives were small in capacity, making a forensic image outside of a lab environment was relatively straightforward and not time intensive. However, in the last decade, storage sizes have grown to the point that even making a basic forensic image is problematic, let alone processing all the data after the creation of the forensic image. [3]

For example, the website StorageReview.com tested a 6TB Western Digital magnetic hard drive. [4] The maximum bandwidth exhibited by the drive for sequential reads was 214.53 Megabytes per Second (MB/s). At that bandwidth, to read all of the sectors on disk to make the forensic image would take approximately 7.77 hours. There would also need to be a full read-back of the data for hash verification to ensure that the data was read and copied correctly. Waiting on-site for at least 7 hours

for just forensic imaging, without verification and without forensic artifact analysis is not feasible, especially when dealing with multiple systems and multiple drives. If 6 TB is difficult enough to address, as of the end of 2018, major retailers were selling 14 terabyte (TB) hard drives. [5] The days of imaging on-site without artifact analysis are quickly ending due to the explosion of larger hard drives.

2.2 Basic Hardware Imagers

Many forensic examiners working in the digital forensics field, both in law enforcement and in the private sector, use a hardware solution for the creation of a bitstream image. These devices are called hardware duplicators, hardware imaging devices, disk imagers or forensic imagers. Examples of these tools include the Tableau devices TD2u, TD3, TX1, Logicube Falcon-NEO Forensic Imager, MediaClone SuperImager devices. [6] [7] [8] All of these devices support the following features:

1. Basic sector imaging, copying the data sector-by-sector or grouping physical sectors together.
2. Creating a cryptographic (or multiple cryptographic hashes) of the source bitstream while copying the data.
3. Performing a read-back of the written data for computing a verification hash for validating that the data copied is an exact match to the source data.

Based on limited testing and datasheet analysis, none of the devices logically analyze the file system for file cluster boundaries for intelligent grouping of sectors for imaging. Additionally, none of the listed devices performs file analysis because none of the devices inspects any of the logical data as it is being copied. For a 14TB hard

drive sold today, any of these devices would take over 24 hours to image, making this process unfeasible.

2.3 Basic Software Imagers

Before hardware imaging devices were commonplace, disk imaging was performed using primarily Linux tools. The standard used was GNU dd, which is a block I/O tool that when used properly can create a bitstream copy of a source device to a destination device or raw image file. [9] The GNU dd tool does not have any capability of performing any live file reconstruction nor analysis of files – it is simply a block I/O software tool.

Two different software tools have been developed from the GNU dd tool with digital forensics features: dc3dd and dcfldd. dcfldd is an "enhanced version of dd" developed by Nicholas Harbour while at U.S. Department of Defense Computer Forensics Lab (DCFL). [10] The latest version of dcfldd is 1.3.4 released February 12, 2006. dc3dd is a "patch" to the GNU dd program. [11] The latest version is 7.2.646 released April 29, 2016. Both tools perform a simultaneous hash of the entire source disk while creating the bitstream image. However, neither tool provides the ability to do any analysis of the live file data, increasing the time delay between preservation and analysis of data.

2.4 Triage

Various techniques have been adopted to attempt to combat the size problem among digital forensics, the majority of which fall into a form of triage, which

attempts to selectively analyze items prior to imaging. [12] The advantage of this technique is that it can be performed in the field and does not require all systems to be brought back to the lab for analysis only to discover that the system did not contain relevant evidence. The problem with this technique is that it requires analysis prior to preservation to identify systems to preserve and analyze in depth at a later point. This delays the preservation of digital evidence, which could take hours as indicated above.

2.5 Evimetry

The issues surrounding software involved in digital forensics imaging and large drives have been tackled by developers. One of the most promising tools written by Dr. Bradley Schatz is Evimetry. [13] Evimetry allows for triage during imaging, allowing the examiner to indicate the areas of the disk to prioritize to collect and preserve for analysis. This is accomplished by using the Advanced Forensics Format (AFF) version 4, which is an evidence file format that creates a forensic image in a non-sequential manner. While extremely promising, Evimetry only outputs the image to AFF4, which is gaining support among commercial forensic tool manufacturers but is not as widely supported as Expert Witness Format (E01) or raw dd files. This is addressed by using a filesystem bridge developed by Schatz Forensics to allow the image file to be mounted to the examination system to be analyzed using any forensics tool.

However, Evimetry does not address the issue of analyzing every file while creating the bitstream image that SPARTA addresses.

2.6 Sifting Collectors

An alternative method is the creation of a sparse or partial logical image of the evidence as opposed to a full physical bitstream forensic image. The most promising of these techniques is the use of sifting collectors. [14] The problem with this technique is the fact that the entirety of the evidence is not collected nor examined. During a criminal trial, the use of sifting collectors will lead to the argument from the opposing counsel that there is a possibility of exculpatory evidence in the region not collected nor analyzed, leading to potential doubt among the jury in the process of the usage of sifting collectors.

The figure below outlines the high-value areas identified by Grier and Richard III as being important versus those that are not used for forensic analysis.

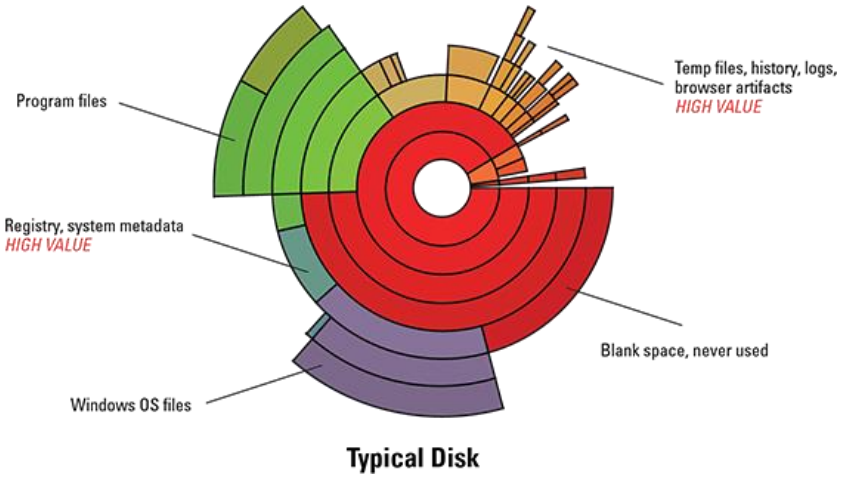


Figure 1 - Breakdown of Typical Disk According to Grier & Richard III

In contrast to Sifting Collectors, SPARTA will analyze all live files, including those identified Windows OS files, in the event that individuals are masquerading files as Windows files.

2.7 LOTA

The most promising area for expediting forensic imaging and analysis is the use of parallel processing: while making the forensic image, perform automated analysis at the file level. This is outlined in a process called a latency-optimized target acquisition (LOTA). [15] However, there are key holes in the paper surrounding the LOTA system that make it unsuitable for a field device for processing all live files during a full physical acquisition.

1. There is a lack of research surrounding the requirements to perform targeted processing of files while performing a bit-stream image of a digital evidence source. The LOTA system details only processing cores required for certain tasks, but does not address memory requirements beyond "assuming a RAM-rich configuration…" [15] In a portable field device that is cost-conscious, a clear establishment of RAM requirements is necessary.
2. The paper clearly identified file fragmentation as an area outside the scope of the project. What is lacking is quantifying the relationship between file fragmentation on-disk with efficacy of parallel processing of logical file data. The LOTA system reference HDD was described as "best case scenario as it was created in one shot." [15] There is no published research literature to analyze the relationship between file processing for digital forensics and file fragmentation, which is something that this thesis seeks to establish.

The SPARTA system, in comparison to the LOTA system, will be a field-capable system that seeks to process all live files while simultaneously creating a full forensic disk image with verification. The results of the live file processing will be saved to a

metadata file for importing into an analysis tool so that efforts do not need to be duplicated after acquisition and partial processing.

2.8 FIREBrick

Portable digital forensic tools are necessary for situations in which the digital source mediums cannot leave the environment. This is common with civil litigation cases where the computers are the property of the opposition and the only authorized work is to make a digital forensic image. Portable open source digital forensic devices have been developed, such as the FIREBrick prototype. [16] Yet when comparing the FIREBrick prototype against the LOTA system, for example, the FIREBrick appears to be woefully underpowered to perform any parallel processing.

2.9 Foundational Related Work

INDXParse is a suite of forensic tools written by Willi Ballenthin, a Reverse Engineer at Mandiant/FireEye. [17] The suite provides a range of capabilities for parsing different data structures unique to the NTFS file system. The prototype for SPARTA will perform the simultaneous analysis of NTFS volumes, and Mr. Ballenthin has made his INDXParse NTFS parser tools available as open source. The project leverages the tools in the suite to parse the master file table records of the drive to build the cluster-to-sector map described in Chapter 3.

The Sleuth Kit (TSK) is a collection of utilities for forensics developed by Dr. Brian Carrier of Basis Technology. [18] The singular usage of TSK in this project is the utility to extract the Master File Table (MFT) from the NTFS formatted volume

prior to creating the bitstream image. To do this, I utilized the following two commands that are a part of TSK: `mmls`, which will list the partition table of a disk and the starting offsets; and `icat`, which will extract a file based on the inode number or MFT entry number. The exact sequence of commands used for extracting the MFT from a given drive is as follows:

```
mmls <source drive> | grep NTFS | awk '{print $3}' | bc
icat -o <sector> <source drive> 0 > MFT.raw
```

The first command will result in an integer number corresponding to the sector number of the NTFS volume. The second command will extract the file corresponding to MFT entry number 0 of the volume starting at the provided sector number of the provided source drive. MFT entry number 0 always refers to the MFT itself, so the result of this command will be the MFT encapsulated as a file for processing.

2.10 CASE/UCO

SPARTA will export all file processing results to an intermediary forensic metadata file utilizing the new Cyber-investigation Analysis Standard Expression (CASE) format. [19] This would be one of the first tools to use the standard that has initial wide support among researchers, the open source community and the large commercial vendors.

CASE is the successor to the intermediary format known as the Digital Forensics Extensible Markup Language (DFXML). [20] It is part of the new Unified Cyber Ontology (UCO), with the CASE portion lead by Dr. Eoghan Casey from the University of Laussane. [21] Dr. Casey and his team had a paper accepted to the

Digital Investigation Journal (Issue 22, September 2017), establishing it as the new standard for tool interchange. Per the website, it has been supported by many companies and vendors, including DC3, FireEye, MITRE, NIST, AccessData, Basis Technology, Blackbag, Cellebrite, Europol, Guidance Software, IBM, Magnet Forensics, NCCoE, Nuix, Oxygen Forensics and the Volatility Foundation.

2.11 Test Dataset Developed

For testing SPARTA, I created a standard lab-built dataset that is fragmented to precise percentages. The dataset will be representative of a real world system: the extracted files from a fully functional Windows 10 system with the entire Govdocs1 corpus data, approximately 1 million files made available through the Digital Corpora. [22] This drive is approximately 282 GB of data on a 320 GB magnetic hard drive. This is in contrast to the testing done in the LOTA system: an ext4 formatted volume with approximately 1.8 million files and the m57 drive from the Digital Corpora, which was a 10 GB virtual disk.

2.12 Fragmentation Generation Tools

To create fragmented disks for testing, I used the tools provided by the company Raxco software as a part of the PerfectDisk tools for artificially creating disk fragmentation. The specific tool I used was called SCRAMBLE.exe. [23] This tool will take a provided disk volume and proceed to implement a pseudo-random fragmentation algorithm to a non-deterministic number of files on the disk. I then tested the fragmentation percentage using the built-in Windows Disk Defragmentation

tool for Windows 10. Should the fragmentation percentage exceed the target, I used the Piriform Defraggler tool to selectively defragment a number of files and retest the volume fragmentation to ensure I was in the correct threshold.

This process allowed for five different physical disks containing the exact same data to be fragmented at the following levels: 0%, 5%, 10%, 20% and 50%.

CHAPTER 3

METHODOLOGY AND SYSTEM DEVELOPMENT

3.1 SPARTA Requirements

The overall system for the SPARTA prototype is a system that will take in as an input a full disk or disk image with a single NTFS volume. NTFS is the focused file system for two reasons: it is designed with a singular structure defining all file cluster allocation (the Master File Table) and it is the default file system for all Windows systems. As output, the prototype will produce the following: a bitstream image file (dd) containing the entire contents of the input disk and a CASE/UCO metadata file with the results of the live file forensic analysis.

3.2 Software Design and Development

After performing the background research, I decided to write the project in Python 2.7. While Python is an interpreted language with known performance limitations, the theory was that with modern processing the limitations of the interpreted scripting/programming language would not be as much of a bottleneck as transfer speeds from the SATA bus. The test results described in Chapter 4 confirmed the theory.

Additionally, the additional components of the CASE/UCO implementation as well as the INDXParse library indicated that the project implementation would be expedited by using Python to align with the existing libraries.

The development environment was a licensed academic version of PyCharm by JetBrains.

3.3 System Design Flow and Pseudocode

While most digital forensic imagers begin immediately reading the bitstream from the source disk, SPARTA requires some preprocessing to process these files live. The preprocessing steps are as follows

1. Read the file signatures list that will be used to validate the file signature with the extension.
2. Examine the source disk and extract the Master File Table from the NTFS volume.
3. Iterate through the Master File Table and analyze each entry for live, allocated file entries.
4. For each live, allocated file entry
    a. If the file has resident data in the Master File Table, immediately queue the file data for processing. Files that are resident to the Master File Table do not use clusters to store data. Rather, the data resides in the Master File Table entry, so the data for the file has already been copied and ready to be processed.
    b. If the file has non-resident data, process the cluster-runs which contain the list of clusters that contain the data for the file.
    c. Map out each cluster run from a logical offset from the previous run (as stored in the Master File Table) to a physical offset from the beginning

of the volume. This data will be stored in the data structure called a cluster map.

   d. Determine which cluster run is the last in the physical ordering and mark it as the last cluster. This will allow the system to know if all clusters for a given file have been read from the bitstream.

5. End of preprocessing for SPARTA

After preprocessing completes, the main portion of the system begins to execute. The main thread begins to read the bitstream image from the source drive. The read for the preliminary portions of the drive, before the NTFS volume, are done sector-by-sector (512 bytes at a time) and written subsequently to the destination drive. While the sectors are being read in individually, the MD5 and SHA1 imaging hashes are being computed.

Once the bitstream read pointer reaches the first sector of the NTFS volume (the Volume Boot Record), the logic proceeds with the main SPARTA parallel processing engine as follows:

1. Create a thread-safe queue called unprocessed file queue.
   a. Create a thread pool for processing these file objects. Initial testing created ten threads in the pool.
2. Create a thread-safe queue called processed file queue.
   a. Create a thread pool for these processed files. Initial testing created one thread in this pool.
3. Set the cluster number to 0.

4. Lookup the current cluster number in the cluster map built from the Master File Table.
5. If the cluster number does not appear in the map, then read the cluster, compute the stream hash, and write the cluster to the bitstream output.
6. If the cluster is in the map
    a. Based on the data in the cluster map, read the entire cluster run length, which is avariable number of consecutive clusters depending on the run.
    b. Add the data block read-in to the correct logical offset of the file object.
    c. If this was the last cluster run for the file (in other words, if all logical clusters have been read), then add the file object to the queue created in step one.
    d. If this was not the last cluster run, then mark in the cluster map that the file object has been created and waiting for the rest of the data.
    e. Write the cluster-run data to the output data stream.
7. Move on to the next cluster if we are not at the end of the drive.
8. Once we reach the end of the drive, wait until the file processing object queue is empty before ending the main thread.

While the bitstream is being processed, the thread pool created above works on the file object queue as follows:

1. Wait until an item is in the unprocessed file queue.
2. Hash the logical file data based on the logical file size extracted from the MFT.

a. Currently, the system supports file MD5 hash. SHA1 hash support would be trivial.
b. Iterate through the file signature list and determine if the first bytes match any signature.
    i. If we have a match, assign that file type to the file and break out of the iteration
c. Move the file object from the unprocessed file queue to the processed file queue
d. Wait until we have another entry, and repeat

The processed file queue is designed to output the results of the file processing to the CASE/UCO output standard as follows:

1. Wait until an item is in the processed file queue.
2. Pop the top item off the queue and send it to the CASE/UCO processor. This will write the appropriate JSON structure to conform to the CASE/UCO standard.
3. Wait until we have another entry.

The following diagram illustrates the overall proposed digital forensics imaging process utilized by SPARTA.

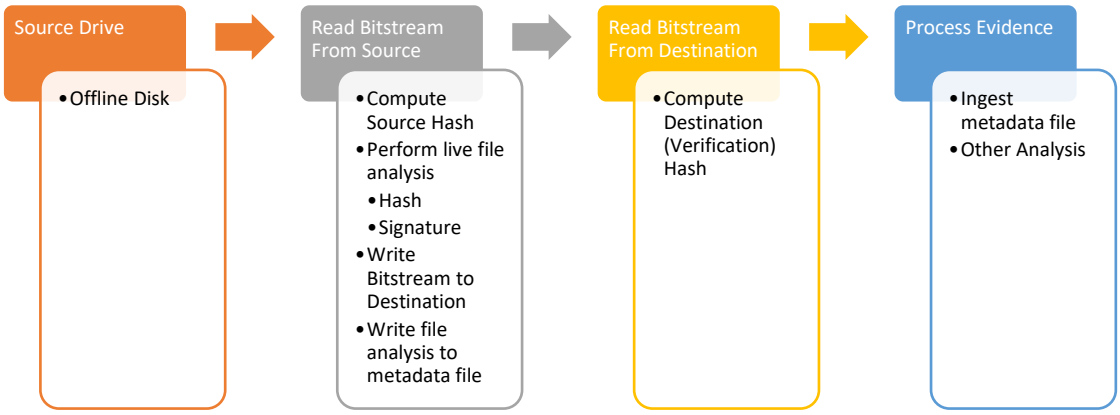


Figure 2 - SPARTA Proposed Digital Forensic Imaging Process

3.4 Fundamental Differences With SPARTA Compared to Other Tools

The following is a comprehensive list of functionality that is unique to SPARTA compared to other industry tools.

1. SPARTA performs an initial pass over the Master File Table (MFT) in the NTFS volume to build an inverse cluster map. This is similar to what is performed in the LOTA system designed by Rousev.
2. SPARTA performs a disk read of variable length based on cluster run size. This is a unique feature only capable because of the initial pass of the MFT. This should allow for faster overall read speeds from the source disk as magnetic media benefits from longer sequential reads. This is a key differentiating factor compared to open source software tools which allow for a fixed-size block for reads but are incapable of variable size based on cluster runs due to the lack of MFT parsing.

   It is unclear if the LOTA system designed by Rousev performs a variable read

based on the cluster size, as his research was solely based upon unfragmented source disks.

3. SPARTA writes the forensic processing results into a CASE/UCO format that will be a standard for data exchange between forensic tools. It is unclear if Wirespeed or LOTA export metadata into any format that can be utilized by other tools.

3.5 Operating System Configuration

The Operating System chosen for the prototype was a Linux operating system. Linux was primarily chosen for both performance and low cost. Linux is a modular system that would allow for a full field unit to remove all unnecessary components from the prototype to maximize efficiency. This is in contrast to consumer operating systems like Microsoft Windows or Apple MacOS, that do not allow for the customization necessary for a performance field unit. Additionally, Microsoft Windows has a retail cost of $129.99 for the Home Edition, which would be an unnecessary cost for the unit.

A requirement of a forensics duplicator/imager is the ability to protect the source evidence from alterations from normal operating system behavior during the forensic imaging. Both Microsoft Windows and Apple MacOS, by default, do not have mechanisms to write-protect directly attached drives. While most Linux distributions by default are not configured to write-protect attached devices, it is a relatively minor change to enable this feature.

The Linux distribution used was Ubuntu 18.04 LTS (long-term support), and within the OS the following changes were made to enable write-protection to attached devices.

1. Using the dconf-editor tool, the following two keys were disabled:

   ```
   org.gnome.desktop.media-handling automount
   org.gnome.desktop.media-handling automount-open
   ```

2. Within the /etc/udev/rules.d file, under 10-myudisks2.rules, the following rule was added:

   ```
   ACTION==”add|change”, SUBSYSTEM==”block”,
   ENV{UDISKS_IGNORE}=”1”
   ```

The two above rules would prevent any attached devices from being automounted as read/write to the Operating System. Attached devices can be mounted as read/write, but would require specific user action to do so. This ensures that the operator would specifically indicate the destination device for all imaging actions. There is the understanding that a user error could result in overwriting of the source (for example, a misused DD command), but this is an assumed risk for all Linux-based forensic imaging devices.

3.6 Hardware Design

The main aspect of SPARTA that differentiates it from many other research projects focusing on parallel processing of digital evidence during forensic imaging is the fact that it is designed as a low-cost field unit. In order to achieve the research goal

of cost-affordability, an investigation into the current hardware availability was necessary to determine if such a goal was attainable.

A selection of motherboard, processor, RAM, video card and solid-state drive (for the Operating System and OS swap space) was necessary for the basic components of the field prototype. Due to the portability requirements of the field prototype, the motherboard would be limited to only Mini-ITX selections. At the time of this research, the latest AMD selection in the Mini-ITX motherboard format was using the AM4 processor socket, while the Intel selection used the 1151 processor socket format. Based on these socket formats, the following processors would be the exemplar processors in the various core configurations needed for testing the performance of the system (Note that for each processor core selection, the processor listed is the lowest cost processor available from Newegg.com, a major retailer of hardware components):

Table 1 - Processor Core Comparison

Number of Cores	Intel Processor	Cost	AMD Processor	AMD Cost
2	Pentium G5500	$94.99	A6-9500	$55.74
4	Core i3-8100	$119.99	Ryzen 3 1200	$94.99
6	Core i5-8400	$193.99	Ryzen 5 1600	$159.99
8	Core i7-9700K	$399.99	Ryzen 1700X	$159.99

Based on the comparison of the major processors available on retail markets, it was decided to build the entire prototype around an AMD platform due to the lower price on all processors with equivalent CPU cores available.

While the amount of RAM available would vary between 8 GB and 32 GB, the type and speed of the RAM would not used as a variable for testing. The RAM chosen would be based on the least expensive configuration available at the time of testing.

The solid-state drive would be used for both the operating system, SPARTA software as well as the operating system virtual memory. To attempt to reduce the impact of the system secondary memory throughput as a performance bottleneck of the system, a selection would be made of a high-speed solid-state drive, preferably of the M.2 variant so that the system would be I/O bound based on the SATA source and SATA destination drives.

Based upon the research above, the following components were used for building and testing the SPARTA prototype as of 6/19/2018.

Table 2 - SPARTA Prototype Complete Component Cost

Component	Make and Model	Cost
Motherboard	ASRock Fatal1ty AB350 Gaming-ITX/ac	$104.99
Processor	AMD Ryzen 7 2700X 8-Core	$319.99
RAM	32GB DDR4 2400	$299.99
Power Supply	Rosewill Hive 550W	$54.99
Case	Cooler Master Elite 130	$40.16
Video Card	Gigabyte Radeon R5 230	$36.99
SSD	Crucial MX500 M.2 2280 1 TB	$219.99
Total		$1,077.10

A unique feature of the motherboard chosen is the ability to enable or disable CPU cores within the BIOS. This enabled testing of the prototype for variable cores to emulate testing different physical CPUs without the requirements of purchasing different CPUs for testing.

The following are images of the completed prototype device.

Figure 3 - Front-facing image of SPARTA Prototype

Figure 4 - Back-facing image of SPARTA Prototype, including SATA connectors.

Note that the backside has two internal SATA and two internal SATA power connectors for the source and destination disks.

CHAPTER 4

TESTING AND FINDINGS

4.1 Development of SPARTA Testing Dataset

In order to establish equivalent testing for the SPARTA prototype as well as existing forensic tools, I developed a dataset that would be representative of a typical user's Windows system. The dataset included files and folder structure used for Windows 10, extracted from a virtual machine provided by the Microsoft Windows Dev Center. [24] Additionally, the files available from the GovDocs corpus from the Digital Corpora were included to give a variety of typical user files. [22] Approximately 410,000 files from the GovDocs were included in the dataset, with data types including Microsoft Word, Excel, PowerPoint, JPEG, PDF and plain text files. The overall dataset of Windows and GovDocs files equaled approximately 282 Gigabytes, making it one of the largest datasets used for testing imaging and processing speeds.

The dataset was copied onto five magnetic hard drives. Each of the drives was a Western Digital Blue Drive, Model WD3200AAKS with 16 Megabytes of cache and 320 Gigabytes in capacity. By ensuring that each of the five drives was the same make and model, I eliminated as much variability in drive mechanics that would impact the performance metrics. All testing of forensic imaging and processing used the same destination disk: a Western Digital Blue WD5000AAKS, 500 Gigabytes in capacity with 16 Megabytes of cache. While each of the drives are only SATA 2 and not SATA

3 speeds, the read and write speeds on average of magnetic media does not exceed the maximum throughput of 3 Gigabits per Second available in SATA 2.

After ensuring that each of the source drives had the dataset copied, I marked them with a specific measure of fragmentation: 0%, 5%, 10%, 20%, and 50%. Raxco software provides free utilities for generating disk fragmentation, to be used for testing defragmentation tools. [25] By utilizing a combination of the Scramble utility, which performs an entire disk fragmentation and Piriform's Defraggler [26], which allows for file-based defragmentation, I was able to accurately establish the appropriate levels of disk fragmentation on each disk as measured by the Windows 10 built-in tool for defragmenting disks.

4.2 Testing Imaging Tools and SPARTA with No Analysis

For each of the forensic imaging tools, the source drive and destination drives used were the 0% Fragmented drive developed in 4.1 and the standard destination drive listed in 4.1. The available hardware devices available for testing were a Tableau TX-1 and a Tableau TD2u. Both were updated to the latest firmware available from the Tableau Firmware Update v7.29 tool. The software tools used were Guymager and DC3DD, tested on the SPARTA hardware platform with 8 physical cores and 32 GB of RAM.

Each of the tools was configured to make a single DD bitstream images, unsegmented, with MD5 and SHA1 hash computations and verifications. The results of the testing are as follows.

Table 3 - Industry Tool Imaging Times

Tool	Time
Tableau TX-1	1:45
Tableau TD2u	1:44
Guymager	1:43
DC3DD	1:43
Average	1:43

It is interesting to note that each of the imaging results are within 2 minutes of each other and the average of 1 hour and 43 minutes.

The SPARTA prototype was tested as a simple forensic imager with no file analysis. The prototype was configured at the maximum specifications of 8 cores and 32 GB of RAM. Three test runs of operating as a simple disk imager were performed, and the results recorded. The average time for SPARTA to perform as a disk imager was 1 hour, 43 minutes, equaling those of the simple imaging software and hardware tools. This result demonstrates that the decision to use Python as the base programming language did not increase the time necessary to perform forensic imaging as compared to established tools. There was a concern that using a language like Python, which is not as efficient as C or C++, would increase the time to image, but the results clearly show that not to be the case.

4.3 Testing Processing Tools

Many of the processing tools for forensics available and widely used in industry are closed-source paid products. I have valid professional licenses for the following tools that I used to test evidence processing: EnCase v. 6.19.7 by OpenText, Forensic Explorer v4.3.5 by GetData, and X-Ways Forensics 19.8 SR-6 by X-Ways Software Technology. Autopsy is an open-source digital forensic analysis tool. Each of the

forensic tools is installed on my forensic workstation with an Intel i7-6700 4-core processor with 64 GB of RAM.

I made a valid forensic image from the 0% fragmented drive as tested using MD5 and SHA1 hash verification and copied the DD image file to the destination drive (WD5000AAKS). I used each of the tools to perform an MD5 hash of each file as well as perform a file signature analysis, which compares the data in the file header with the extension listed in the file name. These are two common forensic practices which many, if not most, examiners perform with each system.

I repeated the file signature and file hash analysis with each level of fragmentation (0%, 5%, 10%, 20% and 50%) from the destination drive. The results are below, with fragmentation level indicated in percentages and time listed as hh:mm.

Table 4 - Industry Tool Analysis Times

Tool	**0%**	**5%**	**10%**	**20%**	**50%**
Autopsy	1:03	1:30	2:55	3:30	2:11
EnCase	0:57	1:15	1:14	1:33	1:48
Forensic Explorer	1:22	2:02	2:27	2:49	3:17
X-Ways	1:03	1:28	2:10	2:39	1:51
Average	1:06	1:33	2:11	2:37	2:25

For better representation, the following chart demonstrates the increase in processing time necessary for each increased level of fragmentation.

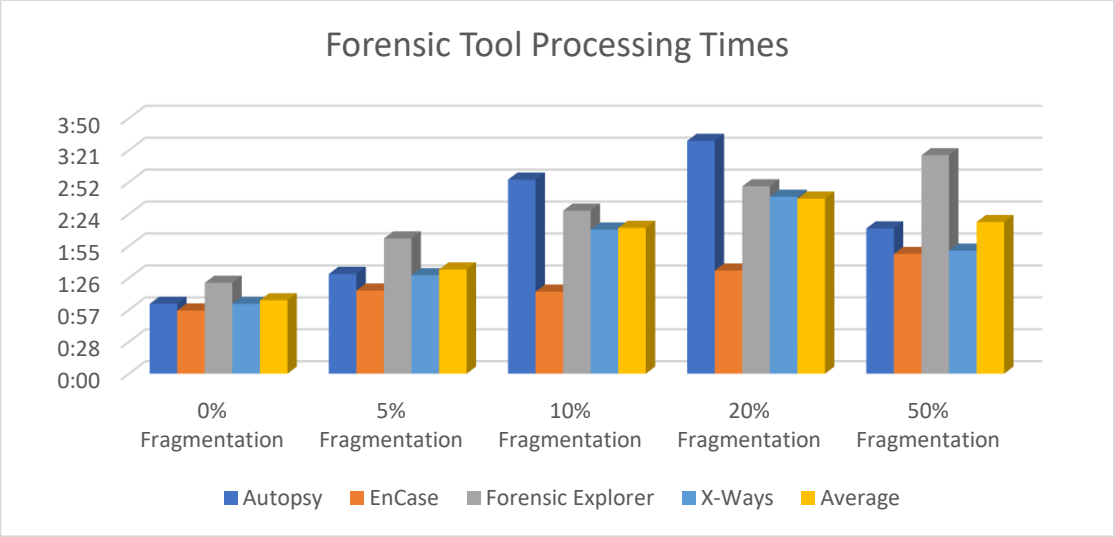


Figure 5 - Forensic Tool Processing Times

With the exception of 10% fragmentation with EnCase and 50% fragmentation with Autopsy, each of the tests performed with an increase in fragmentation led to an increase in the time necessary to process the evidence. Further analysis would need to be performed to determine the rate of increase for processing time compared to fragmentation as no apparent linear or exponential pattern appears to fit well with the data.

4.4 Combined Imaging and Processing Times

Taking the average tool imaging time of 1 hour, 43 minutes with the average processing time for each measured level of fragmentation, we arrive at the following table, which compares the sum of imaging and processing with measured fragmentation percentage.

Table 5 - Combined Average Imaging and Processing Times

Fragmentation Percentage	Time
0%	2:50
5%	3:17
10%	3:55
20%	4:21
50%	4:09

The following measurements are charted below:

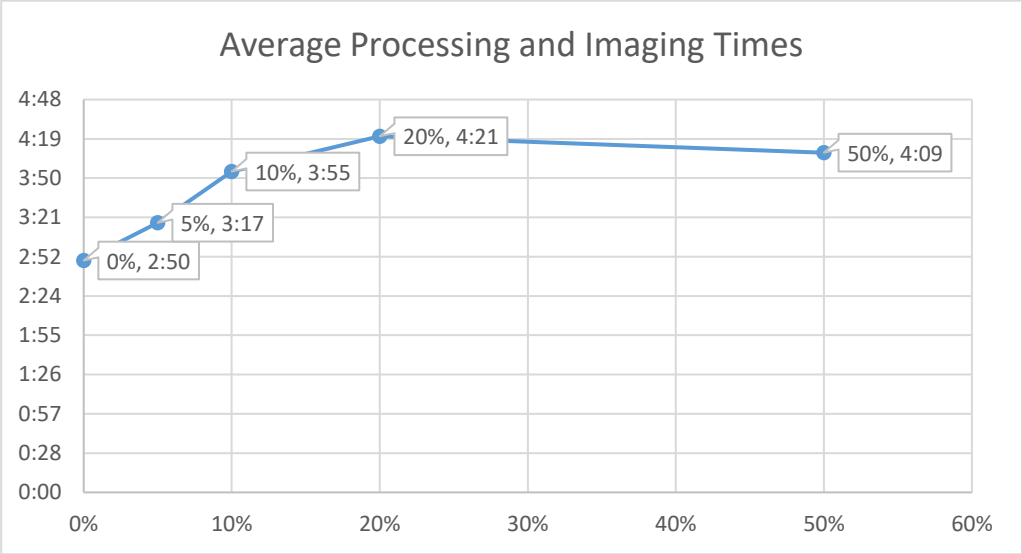


Figure 6 - Combined Average Imaging and Processing Times

We now have the baseline time measurements to determine if the SPARTA design, code and hardware can meet the efficiency standard of being within 10% of industry standard tools based on the averages taken above.

4.5 Testing the SPARTA Prototype With Variable Cores and RAM

The SPARTA prototype was used to test each scenario with varying disk fragmentation percentages of 0%, 5%, 10%, 20% and 50%, varying processing cores of 2, 4, 6 and 8, and varying available system RAM of 8 GB, 16 GB and 32 GB. Each scenario was run three times, with the averages of the three runs documented in the table below.

Table 6 - Summary of SPARTA Testing Results

Test Case	Cores	RAM (GB)	Fragmentation	Mean Time Taken	Mean Major I/O Page Faults
01a	8	32	0%	2:15:49	28
01b	8	32	5%	2:17:24	1032722
01c	8	32	10%	2:40:51	5122570
01d	8	32	20%	2:58:32	8425078
01e	8	32	50%	3:24:21	15262436
02a	6	32	0%	2:15:51	34
02b	6	32	5%	2:19:01	1132811
02c	6	32	10%	2:42:05	5103129
02d	6	32	20%	3:00:12	8477996
02e	6	32	50%	3:22:59	15290330
03a	4	32	0%	2:16:39	30
03b	4	32	5%	2:19:09	1076391
03c	4	32	10%	2:42:29	5125908
03d	4	32	20%	3:01:06	8423466
03e	4	32	50%	3:32:49	15287473
04a	2	32	0%	2:15:52	20
04b	2	32	5%	2:18:50	1099323
04c	2	32	10%	2:42:14	5122519
04d	2	32	20%	3:01:40	8568607
04e	2	32	50%	3:27:04	15212920
05a	8	16	0%	2:22:58	1174263
05b	8	16	5%	2:37:47	9038080
05c	8	16	10%	2:59:24	9318398
05d	8	16	20%	3:16:19	13737885
05e	8	16	50%	3:39:54	19901736
06a	6	16	0%	2:17:37	1150437
06b	6	16	5%	2:34:41	5736121
06c	6	16	10%	2:59:23	9271466
06d	6	16	20%	3:14:31	13621883
06e	6	16	50%	3:47:31	20040071
07a	4	16	0%	2:17:55	1182597
07b	4	16	5%	2:38:35	5728819
07c	4	16	10%	2:58:49	9222047
07d	4	16	20%	3:17:23	13699034
07e	4	16	50%	3:50:05	19827439
08a	2	16	0%	2:18:04	1165167
08b	2	16	5%	2:35:44	5727463
08c	2	16	10%	2:58:58	9325907
08d	2	16	20%	3:16:43	13420610
08e	2	16	50%	3:57:44	19897716

09a	8	8	0%	2:58:54	15676195
09b	8	8	5%	3:12:02	19554098
09c	8	8	10%	3:44:44	27114027
09d	8	8	20%	4:02:46	31095330
09e	8	8	50%	4:43:03	37746095
10a	6	8	0%	2:57:39	15425255
10b	6	8	5%	3:07:12	19677461
10c	6	8	10%	3:44:24	26873317
10d	6	8	20%	4:05:12	31386859
10e	6	8	50%	4:38:09	37996071
11a	4	8	0%	2:56:09	15265835
11b	4	8	5%	3:12:12	19890188
11c	4	8	10%	3:42:36	26682159
11d	4	8	20%	4:09:33	31495900
11e	4	8	50%	4:46:57	37574447
12a	2	8	0%	2:57:23	15212506
12b	2	8	5%	3:10:29	19585107
12c	2	8	10%	3:43:35	26329441
12d	2	8	20%	4:00:50	31200088
12e	2	8	50%	4:37:41	36562924

The time listed in the scenario above is the total time to perform the bitstream imaging, verification as well as perform the full file hash computations and file signature analysis.

4.6 Analysis of Memory Dependency

The table above seems to indicate a fairly significant reliance upon RAM for performing simultaneous file processing, with major page faults being the easiest indicator. Each scenario 01a, 02a, 03a and 04a that had 32 GB of system memory available had Major I/O page faults at fewer than 50. This is in stark contrast to lowering the available RAM to 16 GB, which even in test case 05a with 8 cores led to 1174263 page faults, a significant increase. This indicates that a significant amount of time was spent fetching memory pages from the swap space to populate main

memory, and that for the most efficient simultaneous file processing available RAM should be maximized. Even though there were over one million more page faults, it is likely that the swap space residing on an SSD minimized the time impact to only a 7 minute average increase.

When analyzing the processing speed of the SPARTA imaging and processing and comparing all cases for which 8 processing cores are present, we get the following chart comparing times with fragmentation and RAM.

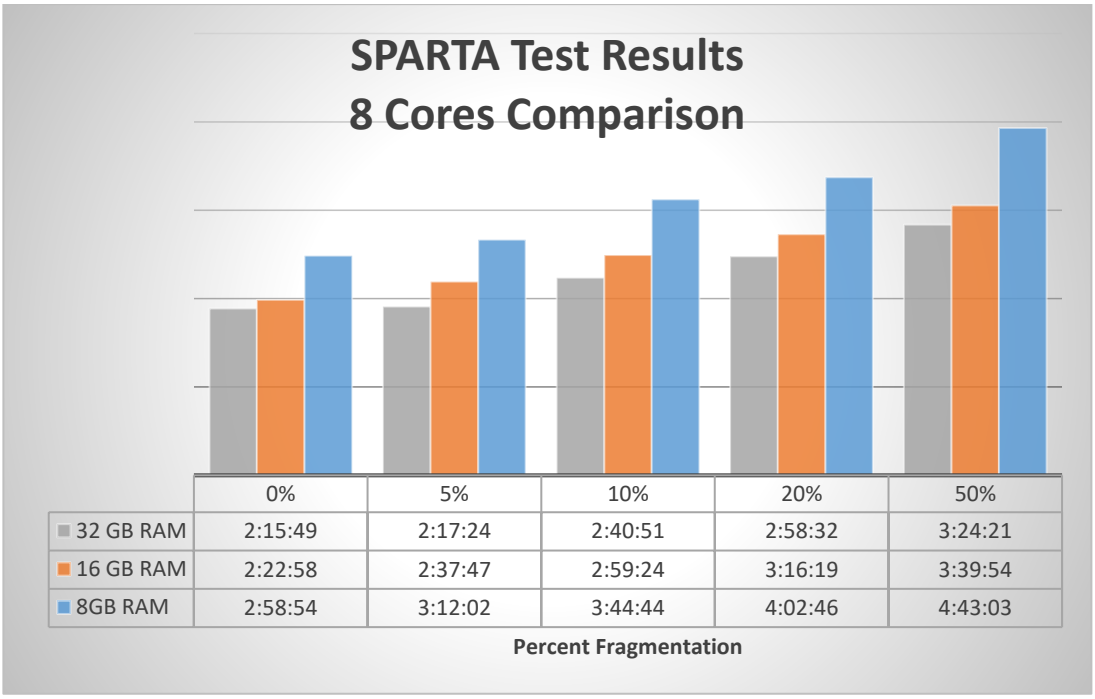


Figure 7 - SPARTA Test Results with 8 Cores

The results indicate a direct relationship between available RAM and processing times, with the dependency becoming more accentuated as fragmentation increases as shown below.

Table 7 - Speed Increases with Increase in RAM With 8 Cores

Fragmentation	Difference Between 16 GB and 32 GB	Speed Increase	Difference Between 8 GB and 32 GB	Speed Increase
0%	0:05:37	4%	0:43:06	24%
5%	0:17:23	11%	0:55:25	29%
10%	0:19:20	11%	1:03:03	28%
20%	0:18:31	9%	1:03:28	26%
50%	0:15:10	7%	1:25:45	30%
	Average	8%	Average	27%

The table above indicates that an average of approximately 8% increases in speed when averaging all runs when increasing the available RAM from 16 GB to 32 GB of RAM. However, when increasing from 8 GB to 32 GB of RAM, the average speed increase is approximately 27%.

The results of comparing SPARTA run times with 6 available processing cores, 4 processing cores and 2 processing cores are all listed below along with charts indicating the speed difference between 8 GB of RAM and 32 GB of RAM, along with the difference between 16 GB of RAM and 32 GB of RAM.

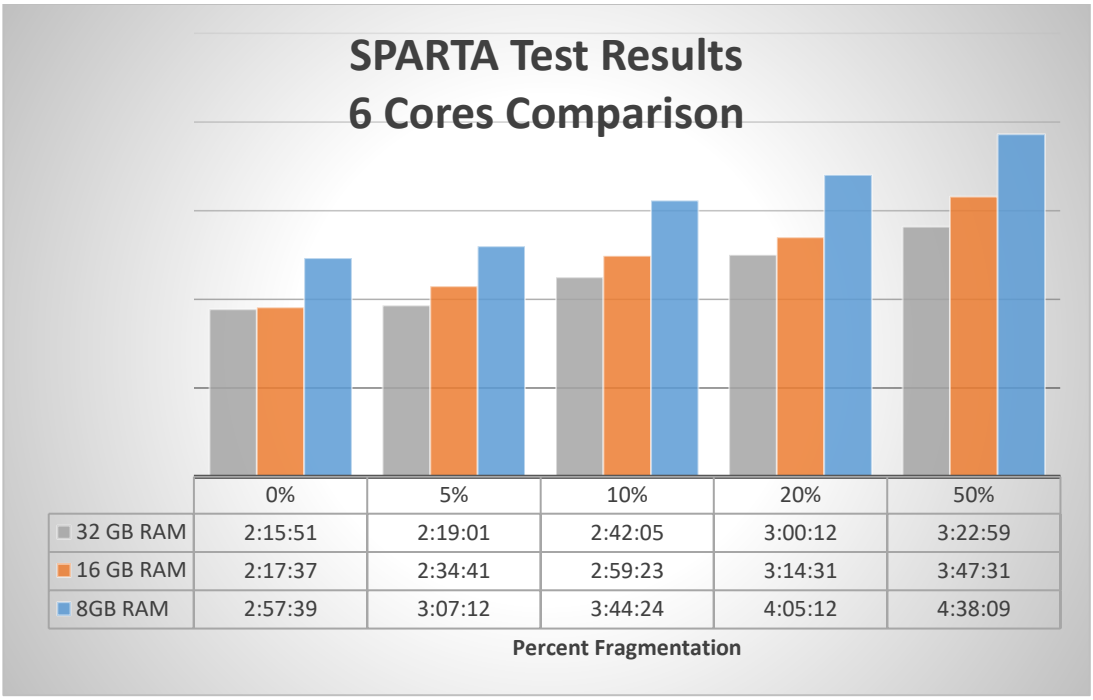


Figure 8 - SPARTA Test Results with 6 Cores

Table 8 - Speed Increases with Increase in RAM With 6 Cores

Fragmentation	Difference Between 16 GB and 32 GB	Speed Increase	Difference Between 8 GB and 32 GB	Speed Increase
0%	0:01:46	1%	0:41:47	23%
5%	0:15:41	10%	0:48:11	25%
10%	0:17:18	10%	1:02:19	28%
20%	0:14:19	7%	1:05:00	27%
50%	0:24:32	11%	1:15:10	26%
	Average	8%	Average	26%

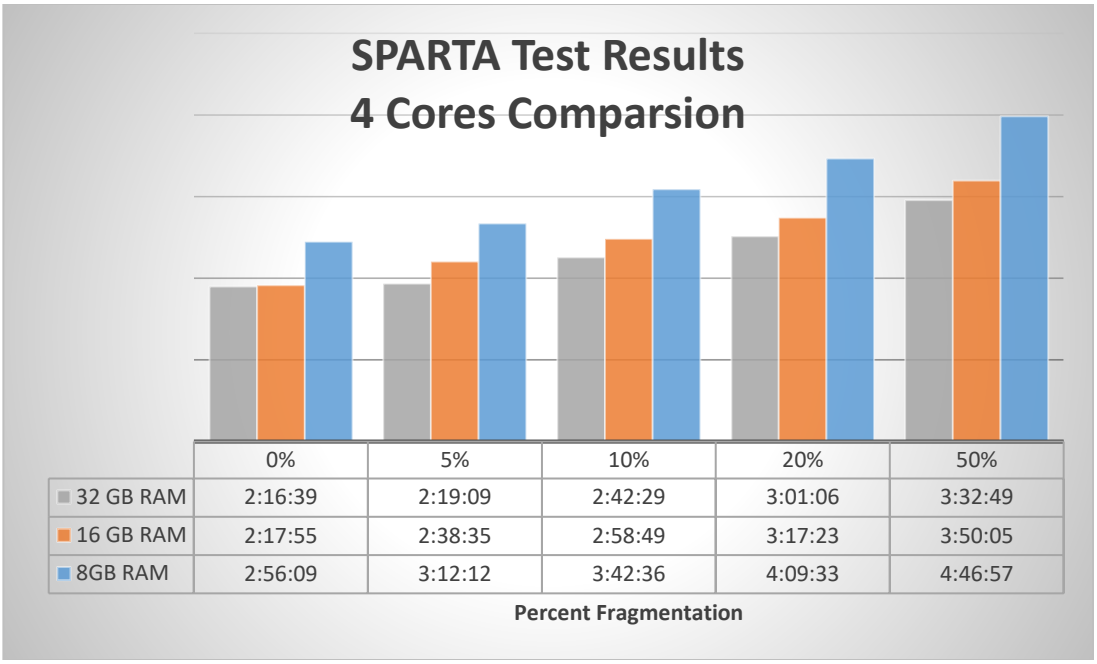


Figure 9 - SPARTA Test Results with 4 Cores

Table 9 - Speed Increases with Increase in RAM With 4 Cores

Fragmentation	Difference Between 16 GB and 32 GB	Speed Increase	Difference Between 8 GB and 32 GB	Speed Increase
0%	0:01:15	1%	0:39:30	22%
5%	0:19:26	13%	0:53:04	28%
10%	0:16:20	9%	1:00:07	27%
20%	0:16:17	8%	1:08:27	28%
50%	0:17:15	8%	1:14:07	26%
	Average	8%	Average	26%

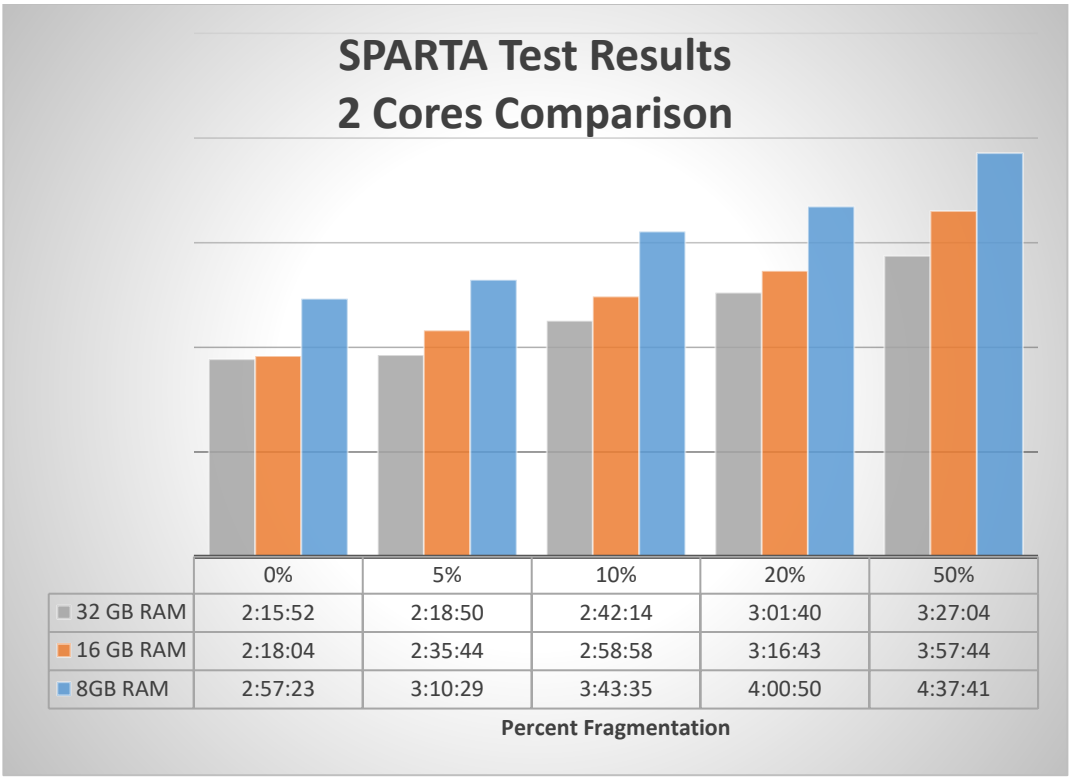


Figure 10 - SPARTA Test Results with 2 Cores

Table 10 - Speed Increases with Increase in RAM With 2 Cores

Fragmentation	Difference Between 16 GB and 32 GB	Speed Increase	Difference Between 8 GB and 32 GB	Speed Increase
0%	0:02:12	2%	0:41:31	23%
5%	0:16:54	11%	0:51:39	27%
10%	0:16:44	9%	1:01:22	27%
20%	0:15:03	8%	0:59:10	24%
50%	0:30:39	14%	1:10:37	24%
	Average	9%	Average	25%

The results of comparing the effects of RAM while keeping processing cores constant provides a nearly identical speed percentage increase relationship. The speed increases were between 7-9% when comparing the results of changing the RAM from 16 GB to 32 GB of RAM, regardless of how many processing cores are available. The speed increases were between 25-27% on average when changing the amount of RAM

from 8 GB to 32 GB of RAM. What is interesting is that the speed increases were more constant when going from 8 GB to 32 GB of RAM, staying between 22% to 27% as opposed to going from 16 GB of RAM to 32 GB of RAM, where the speed increases can be as low as 2% and as high as 14%.

4.7 Analysis of Processing Core Dependency

Table 6 above does not seem to reflect a significant change in processing time when comparing the effects of varying the processing cores. The chart below lists all results when processing the drives using a fixed 32 GB of RAM.

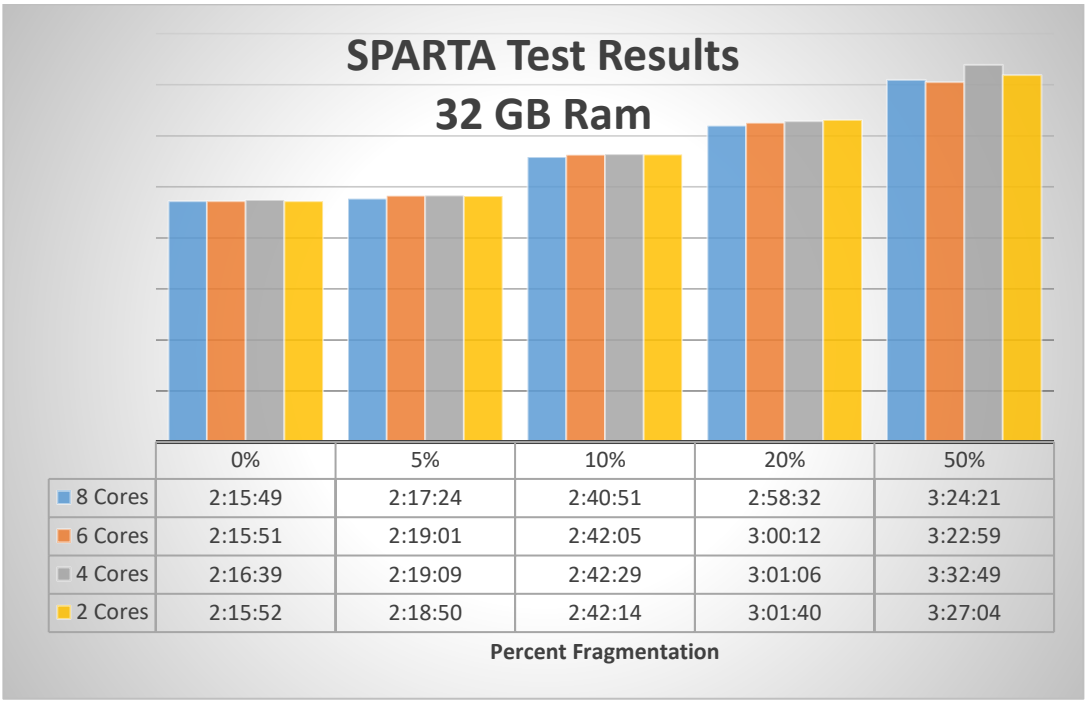


Figure 11 - SPARTA Test Results with 32 GB RAM

We can see that when there is a fixed amount of RAM at 32 GB, the differences between the slowest time for 0% fragmentation at 2:16:39 and the fastest time at 0% fragmentation at 2:15:49 is only 50 seconds, which is negligible at less than 1% compared to the total imaging and processing time is approximately 2 hours and 16

minutes. Even at the 50% fragmentation, the difference between the slowest time of 3:22:59 and 3:32:49 is 9 minutes and 50 seconds, which is approximately 4.8%. However, this difference seems to be exaggerated by the high amount of disk fragmentation at 50%.

Another interesting observation is that the greater the fragmentation, generally the greater the differences in time between the shortest and fastest processing. At 0%, the difference is 50 seconds; at 5%, the difference is 1 minute, 45 seconds; at 10%, the difference is 1 minutes, 38 seconds; at 20%, the difference is 2 minutes, 34 seconds; and at 50%, the difference is 9 minutes and 50 seconds.

However, the overall comparison at least for 32 GB of RAM suggests that fragmentation has a near linear effect on processing time as shown below.

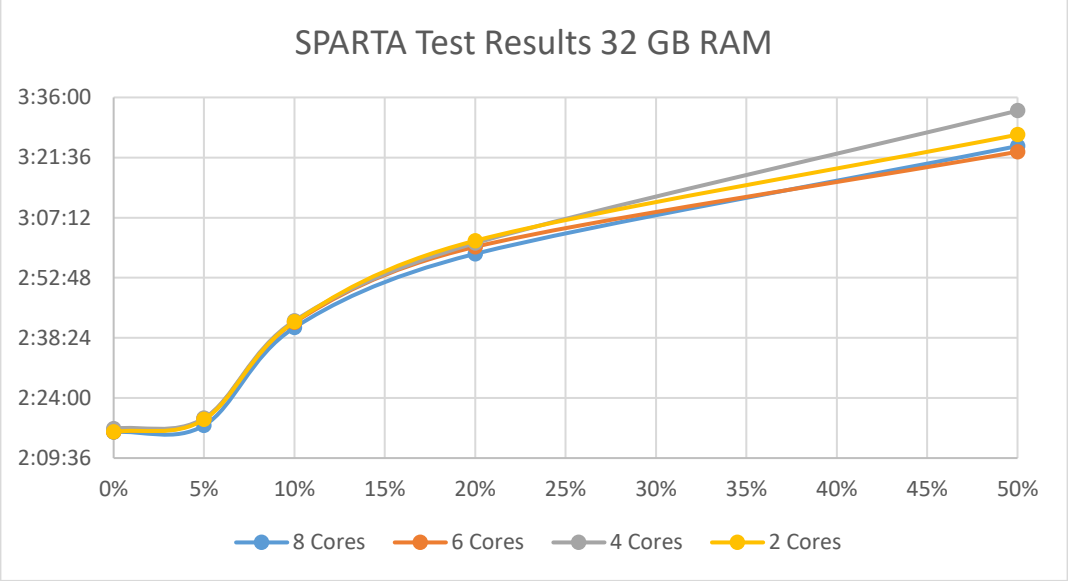
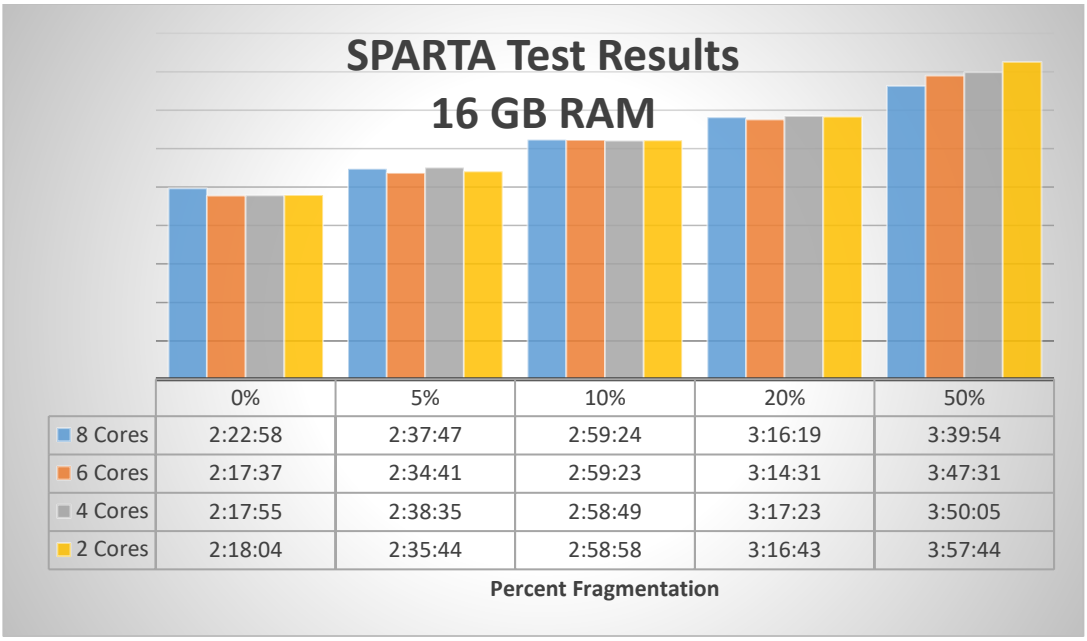


Figure 12 – Fragmentation Effects on Processing Speed with 32 GB RAM

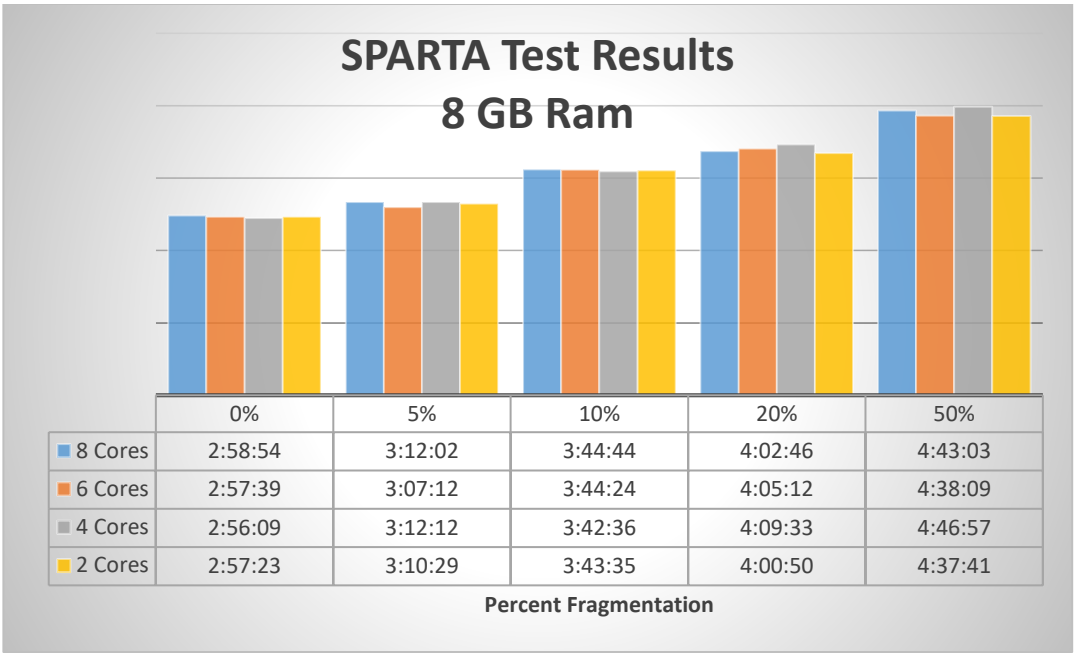
The results from evaluating the performance of SPARTA given 16 GB of RAM largely reflect the results from 32 GB of RAM.



	0%	5%	10%	20%	50%
8 Cores	2:22:58	2:37:47	2:59:24	3:16:19	3:39:54
6 Cores	2:17:37	2:34:41	2:59:23	3:14:31	3:47:31
4 Cores	2:17:55	2:38:35	2:58:49	3:17:23	3:50:05
2 Cores	2:18:04	2:35:44	2:58:58	3:16:43	3:57:44

Figure 13 - SPARTA Test Results with 16 GB RAM

We again see that the difference between the most and fewest cores at 0% fragmentation is 4 minutes, 54 seconds, but it is curious that the 2 Core version performed better than the 8 core version. However, at 50% fragmentation the pattern matches the previous runs with the 8 core version performing 17 minutes and 50 seconds faster than the 2 core version, which equates to a 7.5% speed increase. The results from testing SPARTA with 8 GB of RAM is largely the same as testing it with 16 and 32, with the data demonstrating little variation in testing times between 2 and 8 cores and a linear increase in time to process based on fragmentation.



	0%	5%	10%	20%	50%
8 Cores	2:58:54	3:12:02	3:44:44	4:02:46	4:43:03
6 Cores	2:57:39	3:07:12	3:44:24	4:05:12	4:38:09
4 Cores	2:56:09	3:12:12	3:42:36	4:09:33	4:46:57
2 Cores	2:57:23	3:10:29	3:43:35	4:00:50	4:37:41

Figure 14 - SPARTA Test Results with 8 GB RAM

It is interesting to note that for each of the test scenarios, having 50% fragmentation produced the greatest variability of times to process, even within the testing groups.

4.8 Analysis Of SPARTA Correctness Goal

There are three different aspects of SPARTA that needed to be tested for correctness:

1) The disk image being created. This would be tested using the created MD5 hash over the entire bitstream and comparing that hash to industry tools.
2) The file hashes being created. This would be tested by using the tool output and comparing it with industry tools.

3) The file signatures being created. This would be tested by comparing the signatures identified with the base file types, as none of the files had renamed file extensions to create a mismatch with file signatures.

4.9 Full Disk Imaging Correctness Analysis

The comparison was done using the SPARTA 0% fragmented disk, as all testing for industry tools were performed using this drive.

The Tableau TX-1, TD2u, dc3dd and Guymager all reported the drive and created image file to have the following hashes: MD5 - 92ba9cf58f755ec346eef3806771c96c; SHA1 - 84ef8b1962c3aae4b8fce032f9a4627f6f4b8086.

The first SPARTA test was with no file processing. The log indicated that the source MD5 hash was 92ba9cf58f755ec346eef3806771c96c and the destination (image file) created hash was a match at 92ba9cf58f755ec346eef3806771c96c.

The second SPARTA test was with file processing with 0% fragmentation. The log indicated that the test run with full file processing still generated a disk image with consistent hashes of 92ba9cf58f755ec346eef3806771c96c for MD5 hashes. Each test configuration with varying cores and RAM generated the same disk image with the same MD5 hash, showing correctness with creating disk images.

4.10 File Hash and Signature Correctness Analysis

For testing file hash correctness, each of the fragmentation variations CASE output files were saved for analysis. Ten files were chosen from this group to analyze their signatures and hashes to determine if they were computed correctly. For the hashes, they were compared against an analysis performed by X-Ways, a well-known forensic

tool. For the signatures, they were compared against the extensions of the files since the files did not have a mismatch between the extension and the file signature. The following files were compared and extracted:

Table 11 - Sampling of File Extensions and Hashes

File	Extension	X-Ways MD5 Hash
015153	DOC	0BAD84C28015926331F6D9294B4D015D
099482	XLS	19027D87B59EAAE81C111B852E331184
362288	PPT	7CFF13EC9C8DEA5492DF7A4A58D96C93
438880	JPG	20D8B7B143E76D938D43C50698F8107D
579373	CSV	9D94F7B0628FCFF44B44BD73A4085DE8
624122	LOG	BDD1A7FF781233FDD4189730EBF872F4
673684	TXT	26E4866FBAF23E961C0DEC208B202D84
684368	SWF	592654273948A1CC4C8B810B1DA0C9AD
788215	JPG	5B3971CB3E6B66D023C90091DCCCD8CA
827646	DOC	73E83DCDEEA4DB48A83EEFFFEF856D27

After performing the SPARTA tests on all five levels of fragmentation, the file list above was analyzed for extension and MD5 match. All five test cases produced the same results shown below, as well as an indication as to whether it compared the hash and/or signature correctly.

Table 12 - SPARTA Signature and Hash Analysis Results

File Number	Extension	Signature	Match	Hash	Match
015153	DOC	Microsoft Office Document	Yes	0bad84c28015926331f6d9294b4d015d	Yes
099482	XLS	Microsoft Office Document	Yes	19027d87b59eaae81c111b852e331184	Yes
362288	PPT	Microsoft Office Document	Yes	7cff13ec9c8dea5492df7a4a58d96c93	Yes
438880	JPG	Windows Executable	No	20d8b7b143e76d938d43c50698f8107d	Yes
579373	CSV	None	Yes	9d94f7b0628fcff44b44bd73a4085de8	Yes
624122	LOG	Advanced Stream Redirector	No	bdd1a7ff781233fdd4189730ebf872f4	Yes

673684	TXT	None	Yes	26e4866fbaf23e961c0dec208b202d84	Yes
684368	SWF	Shockwave Flash player	Yes	592654273948a1cc4c8b810b1da0c9ad	Yes
788215	JPG	Windows Executable	No	5b3971cb3e6b66d023c90091dcccd8ca	Yes
827646	DOC	Microsoft Office Document	Yes	73e83dcdeea4db48a83eefffef856d27	Yes

Each of the file hashes were calculated and show that the file reassembly is performed correctly. The signature for 6 of the 10 files were computed correctly, specifically for any Microsoft Office Document or for plain text files, including comma separated values. It appears as though the JPG files were not signature matched correctly and the LOG file was not signature matched correctly. This would indicate an error in the signature match lookup functionality of the tool or an error in the signature tables used. However, this is a slight error that can be corrected in future iterations or production systems. What is more important is the full data reassembly being performed correctly as indicated by the valid hash match.

4.11 Analysis of the SPARTA Efficiency Goal

One of the goals of the SPARTA research is to demonstrate that parallel forensic imaging and processing can be performed faster than sequentially imaging, verifying and then processing the data. The question is further refined due to the previous observation that disk fragmentation has a direct impact on processing speeds, even when only processing evidence after imaging. So each level of fragmentation is analyzed independently and configurations for which SPARTA is faster in all scenarios will be determined to match the efficiency goal.

The following table demonstrates all configuration times at 0% fragmentation as well as the average industry speeds.

Table 13 - SPARTA Speed Versus Industry Mean Speeds With 0% Fragmentation

Test Case	Cores	RAM (GB)	Fragmentation	Mean Time Taken
01a	8	32	0%	2:15:49
02a	6	32	0%	2:15:51
04a	2	32	0%	2:15:52
03a	4	32	0%	2:16:39
06a	6	16	0%	2:17:37
07a	4	16	0%	2:17:55
08a	2	16	0%	2:18:04
05a	8	16	0%	2:22:58
Industry			**0%**	**2:50:00**
11a	4	8	0%	2:56:09
12a	2	8	0%	2:57:23
10a	6	8	0%	2:57:39
09a	8	8	0%	2:58:54

As seen above, every configuration with 16 GB of RAM or greater will beat industry standard averages.

The following table demonstrates 5% fragmentation.

Table 14 - SPARTA Speed Versus Industry Mean Speeds With 5% Fragmentation

Test Case	Cores	RAM (GB)	Fragmentation	Mean Time Taken
01b	8	32	5%	2:17:24
04b	2	32	5%	2:18:50
02b	6	32	5%	2:19:01
03b	4	32	5%	2:19:09
06b	6	16	5%	2:34:41
08b	2	16	5%	2:35:44
05b	8	16	5%	2:37:47
07b	4	16	5%	2:38:35
10b	6	8	5%	3:07:12
12b	2	8	5%	3:10:29
09b	8	8	5%	3:12:02
11b	4	8	5%	3:12:12

Industry	**5%**	**3:17:00**

It is interesting to note that at 5% fragmentation, every scenario will beat industry standard tools.

The following table demonstrates 10% fragmentation.

Table 15 - SPARTA Speed Versus Industry Mean Speeds With 10% Fragmentation

Test Case	Cores	RAM (GB)	Fragmentation	Mean Time Taken
01c	8	32	10%	2:40:51
02c	6	32	10%	2:42:05
03c	4	32	10%	2:42:29
04c	2	32	10%	2:42:14
05c	8	16	10%	2:59:24
06c	6	16	10%	2:59:23
07c	4	16	10%	2:58:49
08c	2	16	10%	2:58:58
09c	8	8	10%	3:44:44
10c	6	8	10%	3:44:24
11c	4	8	10%	3:42:36
12c	2	8	10%	3:43:35
Industry			**10%**	**3:55:00**

Again, every scenario at 10% fragmentation will be faster than industry tools.

The following table is the result of all testing at 20% fragmentation.

Table 16 - SPARTA Speed Versus Industry Mean Speeds With 20% Fragmentation

Test Case	Cores	RAM (GB)	Fragmentation	Mean Time Taken
01d	8	32	20%	2:58:32
02d	6	32	20%	3:00:12
03d	4	32	20%	3:01:06
04d	2	32	20%	3:01:40
05d	8	16	20%	3:16:19
06d	6	16	20%	3:14:31
07d	4	16	20%	3:17:23
08d	2	16	20%	3:16:43
09d	8	8	20%	4:02:46
10d	6	8	20%	4:05:12
11d	4	8	20%	4:09:33
12d	2	8	20%	4:00:50
Industry			**20%**	**4:21:00**

Once again, SPARTA is faster in every configuration at 20% fragmentation compared to industry tools.

Table 17 - SPARTA Speed Versus Industry Mean Speeds With 50% Fragmentation

Test Case	Cores	RAM (GB)	Fragmentation	Mean Time Taken
02e	6	32	50%	3:22:59
01e	8	32	50%	3:24:21
04e	2	32	50%	3:27:04
03e	4	32	50%	3:32:49
05e	8	16	50%	3:39:54
06e	6	16	50%	3:47:31
07e	4	16	50%	3:50:05
08e	2	16	50%	3:57:44
Industry			**50%**	**4:00:00**
12e	2	8	50%	4:37:41
10e	6	8	50%	4:38:09
09e	8	8	50%	4:43:03
11e	4	8	50%	4:46:57

When we get to 50% fragmentation, the results are similar to the findings at 0% fragmentation where all configurations in which there is at least 16 GB of RAM, SPARTA will have faster imaging and processing speeds compared to industry tools.

So combining the results of all efficiency metrics establishes that given a simultaneous digital forensics processing system with at least 16 GB of RAM, it will outperform parallel processing using industry standard tools. It is notable that any processing core configuration had no bearing over the ability for the system to reach its efficiency goal.

4.12 Analysis of the SPARTA Cost-Effectiveness Goal

Based on the results of the efficiency goal, we have determined that the minimum specifications for the SPARTA prototype are 16 GB of RAM, with any core configuration. The design outlined in section 3.4 used a 32 GB RAM configuration.

We can alter the configuration slightly to reduce the cost while still keeping the configuration to meet the efficiency goals as follows:

Table 18 - Cost Optimized SPARTA Components

Component	Make and Model	Cost
Motherboard	ASRock Fatal1ty AB350 Gaming-ITX/ac	$104.99
Processor	**AMD Ryzen 7 2400G Quad-Core**	**$138.82**
RAM	**16GB DDR4 3000**	**$77.99**
Power Supply	Rosewill Hive 550W	$54.99
Case	Cooler Master Elite 130	$40.16
Video Card	Gigabyte Radeon R5 230	$36.99
SSD	**Crucial MX500 M.2 2280 1 TB**	**$109.99**
Total		$563.93

Note: prices on SSDs have come down significantly since the beginning of this research project – nearly 50%.

The above component list indicates that the Cost-Effectiveness goal of having commercial off the shelf (COTS) parts being less than $1000 is easily attainable with a total component cost, before tax, equal to $563.93 when queried on September 14, 2019.

4.13 Analysis of the SPARTA Cross-Compatibility Goal

To achieve the cross-compatibility goal, a suitable metadata interchange format was selected, the Cyber-investigation Analysis Standard Expression. The ontology saves the results of forensic analysis into a standard JSON file. A Python API is available on GitHub and was used in the software implementation in SPARTA.

In testing to determine whether the CASE output from SPARTA matches the raw comma-separated values used in determining the correctness, it appeared as though the CASE output was not consistent with the expected outputs. This could be to a

miscoding or may be due to an implementation issue in the API. All file metadata is reflected accurately in the CASE output but the file signature and cryptographic hash information is not accurate. Further testing and debugging with the CASE Python API developers may be necessary, but this would be a fairly easy fix for production systems.

CHAPTER 5

CONCLUSION

The primary goal of the dissertation was to determine if a new process for digital forensics imaging could be developed to allow for simultaneous processing and forensic imaging of magnetic hard drives. Based on my professional experience and discussions with other examiners, I observed that a lot of human time was wasted waiting for a drive to be imaged before any forensic analysis could be performed. To save human time, different strategies have been researched to solve this problem. The strategy I have employed was to determine if enough processing power was available in commercial off the shelf (COTS) parts to allow for a limited set of forensic analysis to be performed over all live files faster than sequentially processing all live files after forensic imaging. The additional research quantification involved determining the limiting factors of being able to perform all live file imaging. Specifically, the research endeavored to determine if available system RAM, processing cores, or disk fragmentation were the major factors impacting system performance to be able to perform simultaneous disk imaging and forensic file processing.

After performing nearly 570 hours of testing (nearly 24 straight days), the conclusions of the research are as follows:

1) Simultaneous processing of all live files during forensic imaging is not only possible but will perform faster than sequentially imaging then performing forensic analysis using industry standard tools.

2) Processing cores has no significant impact over the ability to perform the requisite tasks.
3) Disk fragmentation has a near-linear impact over performance for all types of file analysis, whether performed after the disk has been imaged or during forensic imaging.
4) RAM has the greatest impact on whether forensic file analysis can be performed during disk imaging and the impact appears to be exponential. With only 8 GB of RAM available, file analysis cannot be performed, but with 16 GB or greater, all files can be subjected to a limited set of forensic analysis during bitstream imaging.

The contributions to the research community are significant. The first being the establishing fact that disk fragmentation has a direct impact on the speed for file analysis. This has wide-ranging implications in both research and industry tools. No published research has quantified the effects of disk fragmentation on file processing, and from the test results it is apparent that there is a direct impact.

Another contribution is the creation of fragmented datasets that can be used in the forensic research community. While the dataset base is the Windows Operating System and the Digital Corpora datasets, they are fragmented to precise measurements and can be used for other testing.

Yet another contribution is the research results to indicate that all standalone digital forensic imaging devices used in industry are obsolete. Manufacturers and designers should be designing devices that can perform a selected set of digital forensic file processing tasks while creating the bitstream image. Utilizing the available processing

power of the hardware, so long as enough RAM is provided, will result in faster time for forensic investigators to begin analyzing processed data.

There are a series of limitations in the research project and implementation. The first being the focus on magnetic hard drives. The design of solid state drives would suggest that fragmentation does not play a role in ability to process data since data access is constant across flash memory. However, without comprehensive testing, this remains only a theory.

The second limitation is the selected set of file processing tasks, namely file hash and file signature analysis. There are many other types of forensic file processes that can be performed, including file indexing, compressed file expansion, registry analysis, photo EXIF data analysis, and keyword searching. While Roussev began to outline CPU cores necessary to achieve some of these tasks, further research needs to be performed to determine both CPU cores and RAM requirements to perform different forensic file analysis techniques while factoring in disk fragmentation.

The third limitation is the limitation on the CASE standard. While it has not gained as widespread adoption as I hoped by the end of this research, the list of contributing companies is promising and hopefully as time progresses more tools will allow for importing of CASE data.

A fourth limitation is the base dataset used for testing. The only filesystem tested was NTFS. FAT, APFS, ext and ZFS all have different structures for tracking fragmentation and can lead to differing results in the effects of fragmentation on forensic file analysis.

A potential limitation could be the size of the source disk compared to the internal SSD used for swap space. Should the drive be too big for all of the incompletely analyzed file fragments to be stored in the swap space, the system could crash. A potential remedy is to turn off all simultaneous file processing when the source disk is over double the size of the internal swap space, ensuring that all file fragments could be stored in swap until the complete file is read and removed from the swap space.

The most easily identified future work is to do testing on Solid State Drives similar to what I did in testing the SPARTA prototype. Comprehensive testing of fragmentation on solid state drives (since SSDs support sector-based file systems such as NTFS and FAT) could lead to conclusions on whether or not fragmentation has any bearing on SSD performance. Additionally, testing SSDs will allow forensic examiners to determine if the increased cost of SSDs will lead to more efficient human time.

An additional area of future work would be to apply the same technique to other file systems, such as FAT. Having this ability to perform simultaneous processing of live files while imaging large USB removable disks formatted with FAT32 would be a boon to forensic investigators.

An additional area of future work would be to expand on this research and Roussev's work to determine the full requirements of forensic file processing, to include all types of file processing normally and potentially performed by forensic investigators. This research established that more than CPU cores are variables in efficacy of file processing, but further research can be performed in this area.

This research endeavor began because I had logged too much time waiting for a disk to finish imaging before I could begin with forensic file analysis. I wanted to determine

if, while I was waiting around for the image to be created, if some forensic file analysis could be performed over all live files and if it would save time in the long-run. My research conclusively states that the answer is yes: by giving a system enough RAM, regardless of how much disk fragmentation exists, forensic analysis can be performed while creating the bitstream image and it would save time compared to established processes. I hope that forensic imaging device manufacturers read this paper so that new devices can be made to speed up what we are trying to do: establish truth in a court of law.

APPENDICES

APPENDIX I – SPARTA SOURCE CODE

```
#!/usr/bin/python

##Joseph Greenfield
##jsgreenfield@my.uri.edu
##

import argparse
import hashlib
import time
import datetime

# for multi-threaded
from Queue import Queue
from threading import Thread

#for CASE/UCO Output
import case

#To add the INDXParse library
import sys
sys.path.insert(0, "/home/joe/INDXParse")

#INDXParse stuff
from BinaryParser import Mmap
from BinaryParser import OverrunBufferException
from MFT import *

#Progress Bar
from progressbar import ProgressBar, Percentage, Bar, ETA, AdaptiveETA

# setting up a global Queues for file processing
num_processing_threads = 10
unprocessedFileQueue = Queue()
processedFileQueue = Queue()

# instantiating the output document
case_output = case.Document()

# instantiating the file signatures list
file_signatures = []


# This will replace the tuples stuff that I wrote below
class FileData(object):

    def __init__(self, mft_record):
        self.mft_record = mft_record
```

```
    def mft_record(self):
        return self.mft_record


class UnprocessedFileData(FileData):

    def __init__(self, mft_record, file_data):
        super(UnprocessedFileData, self).__init__(mft_record)
        self.file_data = file_data


class ProcessedFileData(FileData):

    def __init__(self, mft_record, file_hash, file_signature):
        super(ProcessedFileData, self).__init__(mft_record)
        self.file_hash = file_hash
        self.file_signature = file_signature

class ClusterMapEntry(object):

    def __init__(self, run_length, mft_record, file_offset, last_run):
        self.run_length = run_length
        self.mft_record = mft_record
        self.file_offset = file_offset
        self.last_run = last_run

class FileSignatureEntry(object):

    def __init__(self, fileDescription, fileSig, fileExt, fileCategory):
        self.fileDescription = fileDescription
        self.fileSig = fileSig
        self.fileExt = fileExt
        self.fileCategory = fileCategory

def processFileFromQueue():
    while True:
        unprocessedFile = unprocessedFileQueue.get()

        filename =
unprocessedFile.mft_record.filename_information().filename()
        #print("Processing File: {}".format(filename))

        # hash only the logical file size
        filesize = unprocessedFile.mft_record.data_attribute().data_size()
        md5hash = hashlib.md5(unprocessedFile.file_data[0:filesize])

        fileSiganture = ""
        # checking signatures
        for signature in file_signatures:
            signature_length = len(signature[1])
            file_bytes_to_check =
unprocessedFile.file_data[0:signature_length]

            # signature_as_string = base64.encode(signature[1])
```

```
            if (file_bytes_to_check == signature[1]):
                # we have a signature match
                fileSiganture = signature[0]
                break


processedFileQueue.put(ProcessedFileData(unprocessedFile.mft_record,
md5hash, fileSiganture))

        unprocessedFileQueue.task_done()


def writeCaseOutput():
    while True:
        processedFile = processedFileQueue.get()
        fn = processedFile.mft_record.filename_information()
        si = processedFile.mft_record.standard_information()
        data = processedFile.mft_record.data_attribute()

        case_file = case_output.create_uco_object('Trace')
        case_file_property = case_file.create_property_bundle(
            'File',
            fileName=fn.filename(),
            extension=os.path.splitext(fn.filename()),
            isDirectory=False,
            createdTime=si.created_time(),
            accessedTime=si.accessed_time(),
            modifiedTime=si.modified_time(),
            metadataChangeTime=si.changed_time(),
            sizeInBytes=data.data_size()
        )

        #print ("Processed Filename: {}\tHash: {}\tSignature: {}".format(
        #     processedFile.mft_record.filename_information().filename(),
        #     processedFile.file_hash.hexdigest(),
        #     processedFile.file_signature))
        processedFileQueue.task_done()

        #Temporary Debugging


def parseMFTForFiles(mftpath):
    # initializing the physical cluster map
    # the key for the cluster map will be the physical cluster
    # the value will be a tuple [length of run, mft_record for run, logical
offset within the file,
    # and a boolean as to whether it is the last run in the runlist

    MFTProcessStart = datetime.now()
    print ("Beginning processing MFT at {}".format(MFTProcessStart))
    sys.stdout.flush()

    cluster_map = {}
```

```
    with Mmap(mftpath) as mftbuffer:
        enum = MFTEnumerator(mftbuffer)
        num_records = enum.len()

        pbar = ProgressBar(widgets=[
            "MFT Records Processed: ", Percentage(),
            ' ', Bar(),
            ' ', AdaptiveETA(),
        ], maxval=num_records).start()
        for mft_id in range(0, num_records):
            try:
                mft_record = enum.get_record(mft_id)
                if not mft_record.is_directory() and
mft_record.is_active():
                    # the record is a file and allocated
                    # building the clustermap

                    data_attrib = mft_record.data_attribute()
                    filename_attrib = mft_record.filename_information()
                    # if the data is non-resident, then we care. Otherwise,
the data is in the attribute
                    if data_attrib and filename_attrib and
data_attrib.non_resident() > 0:

                        runlist = mft_record.data_attribute().runlist()
                        dataruns = runlist.runs()

                        # The code in MFT.py actually gives the runlist as
volume offsets
                        # This will keep track of where in the logical file
the cluster run should be
                        file_offset = 0
                        last_offset = 0

                        for (offset, length) in dataruns:
                            cluster_map[offset] = ClusterMapEntry(length,
mft_record,file_offset, False)
                            file_offset += length
                            if offset > last_offset:
                                last_offset = offset
                        cluster_map[last_offset].last_run = True
                pbar.update(mft_id + 1)


            except OverrunBufferException:
                return
            except InvalidRecordException:
                mft_id += 1
                continue
        pbar.finish()

    MFTProcessEnd = datetime.now()
    print ("Complete Processing MFT. Time Taken: {}".format(MFTProcessEnd -
MFTProcessStart))
```

```
    return cluster_map


def printClusterMap(cluster_map):
    for cluster in cluster_map:
        cm_entry = cluster_map[cluster]
        print(
            "Cluster: {}\tLength: {}\tOffset: {}\tFile: {}\tLast Cluster:
{}".format(cluster, cm_entry.run_length,

cm_entry.file_offset,

cm_entry.mft_record.filename_information().filename(),

cm_entry.last_run))

def parseMBRforVBRLocation(mbr):
    # grab the first partition entry, and return the starting sector
    return struct.unpack("<I", mbr[454:458])[0]


def parseVBRforSectorsPerCluster(vbr):
    return struct.unpack("B", vbr[13:14])[0]

def parseVBRforTotalSectors(vbr):
    return struct.unpack("<Q", vbr[40:48])[0]


##This is the code for the non-threaded version
# def processFile(mft_record, file_data, file_signatures, case_output):
#     filename = mft_record.filename_information().filename()
#     #hash only the logical file size
#     filesize = mft_record.data_attribute().data_size()
#     md5hash = hashlib.md5(file_data[0:filesize])
#     fileSiganture = ""
#
#     #checking signatures
#     for signature in file_signatures:
#         signature_length = len(signature[1])
#         file_bytes_to_check = file_data[0:signature_length]
#
#         #signature_as_string = base64.encode(signature[1])
#
#         if (file_bytes_to_check == signature[1]):
#             #we have a signature match
#             fileSiganture = signature[0]
#             break
#
#
#     print("File: {}\tMD5 Hash: {}\tSignature: {}".format(filename,
md5hash.hexdigest(),fileSiganture))
#
#     #adding the file to the output
#     case_file = case_output.create_uco_object('Trace')
```

```
#     case_file_property = case_file.create_property_bundle(
#         'File',
#         fileName=filename
#     )

def main():

    parser = argparse.ArgumentParser(description='SPARTA: System for '
                                                 'Parallel Acquisitions
with '
                                                 'Real-Time Analysis')
    parser.add_argument('source', action="store", help="Source Path (Device
or DD Image")
    parser.add_argument('destination', action="store", help="Destination
File Path")
    parser.add_argument('metadata', action="store", help="Path for file
metadata")
    parser.add_argument('mft_path', action="store", help="Source MFT path")
    parser.add_argument('--file_processing', action='store_true',
default=False, dest='file_processing')
    arg_results = parser.parse_args()

    sourcepath = arg_results.source
    destpath = arg_results.destination
    mftpath = arg_results.mft_path
    mpath = arg_results.metadata
    file_processing = arg_results.file_processing

    # writing preliminary information for Case output
    instrument = case_output.create_uco_object(
        'Tool',
        name='SPARTA',
        version='0.1',
        creator='Joseph Greenfield')

    performer = case_output.create_uco_object('Identity')
    performer.create_property_bundle(
        'SimpleName',
        givenName='Joe',
        familyName='Greenfield'
    )

    action = case_output.create_uco_object(
        'ForensicAction',
        startTime=datetime.now()
    )
    action.create_property_bundle(
        'ActionReferences',
        performer = performer,
        instrument=instrument,
        object=None,
        result=[]
    )
```

```
    # instantiating our file processing threads
    for i in range(num_processing_threads):
        t = Thread(target=processFileFromQueue)
        t.daemon = True
        t.start()

    # instantiating our Case output builder thread
    t = Thread(target=writeCaseOutput)
    t.daemon = True
    t.start()

    # building datastructure for file signatures
    # right now, it will iterate through each signature and see if there is
a match
    # this is a very inefficient way to do it, but we'll see if there is a
significant impact on performance
    if file_processing == True:
        with open("signatures_GCK.txt", "r") as signatures:
            for line in signatures:
                currline = line.split(",")
                fileDescription = currline[0]
                fileSig = currline[1].replace(" ", "")
                fileExt = currline[4]
                fileCategory = currline[5].strip('\n')

                # fileSigBytes = fileSig.split(" ")
                # trying to convert the string to a byte array
                fileSigBytes = bytearray.fromhex(fileSig)

                file_signatures.append((fileDescription, fileSigBytes,
fileExt, fileCategory))

        # reading MFT for processing
        cluster_map = parseMFTForFiles(arg_results.mft_path)

    #printClusterMap(cluster_map)

    # we are building a dictionary of files that actually contain the
binary data for each file
    # the key will be the MFT record number, the value will be the binary
data
    files = {}

    # Disk imaging functionality
    with open(arg_results.destination, "wb") as dest:
        # attempting to open the source disk for stream reading
        print ("Destination file {} open for writing".format(destpath))
        md5hash = hashlib.md5()
        # starting timer
        start = time.time()

        source_numbytes = 0
        #determining number of bytes in the input drive
        fd = os.open(arg_results.source, os.O_RDONLY)
```

```
        try:
            source_numbytes = os.lseek(fd, 0, os.SEEK_END)
        finally:
            os.close(fd)

        curr_sector = 0

        with open(arg_results.source, "rb") as source:
            # trying to read 512 byte blocks
            print ("Source file {} open for reading".format(sourcepath))
            # first block is MBR. Parse it.

            if file_processing == False:
                source_numsectors = source_numbytes / 512 + 1
                pbar = ProgressBar(widgets=[
                    "Sectors Read: ", Percentage(),
                    ' ', Bar(),
                    ' ', AdaptiveETA(),
                ], maxval=source_numsectors).start()

                curr_sector = 0
                block = source.read(512)

                while block:
                    curr_sector += 1
                    pbar.update(curr_sector)
                    md5hash.update(block)
                    dest.write(block)
                    block = source.read(512)

            else:
                source_numsectors = source_numbytes / 512 + 1
                pbar = ProgressBar(widgets=[
                    "Clusters Read: ", Percentage(),
                    ' ', Bar(),
                    ' ', AdaptiveETA(),
                ], maxval=source_numsectors).start()

                block = source.read(512)
                curr_sector += 1
                pbar.update(curr_sector)
                vbr_sector = parseMBRforVBRLocation(block)
                md5hash.update(block)
                dest.write(block)

                # Now reading/writing padding sectors until VBR
                block = source.read(vbr_sector * 512 - 512)
                curr_sector = vbr_sector - 1
                pbar.update(curr_sector)
                md5hash.update(block)
                dest.write(block)

                # We should now be at the VBR. We should now be reading the
VBR ($Boot)
```

```
                block = source.read(512)
                # $Boot is cluster number 0
                clusterNum = 0
                curr_sector += 1
                # lookup the entry in the cluster map
                map_entry = cluster_map[clusterNum]
                # update our cluster numbering to the next cluster after
the full run
                clusterNum += map_entry.run_length
                sectors_per_cluster = parseVBRforSectorsPerCluster(block)
                bytes_per_cluster = sectors_per_cluster * 512
                #total_clusters =
parseVBRforTotalSectors(block)/sectors_per_cluster
                #md5hash.update(block)
                #dest.write(block)

                # we now have to read the rest of the $boot file
                block += source.read(bytes_per_cluster *
map_entry.run_length - 512)
                curr_sector += (map_entry.run_length - 1) *
sectors_per_cluster
                md5hash.update(block)
                dest.write(block)

                # if the $boot is done (unfragmented), then process it,
otherwise, we'll move on to the main processing code
                if map_entry.last_run:
                    # (mft_record,block, file_signatures, case_output)

unprocessedFileQueue.put(UnprocessedFileData(map_entry.mft_record, block))
                else:
                    # we add the $boot to the file map
                    files[map_entry.mft_record.mft_record_number] = block


                # we read the rest of the drive by cluster runs
                while block:
                    # if this cluster is assigned to a valid file
                    if clusterNum in cluster_map:
                        #[cluster_run_length, mft_record, offset, last_run]
= cluster_map[clusterNum]
                        map_entry = cluster_map[clusterNum]

                        mft_record_num =
map_entry.mft_record.mft_record_number()
                        # read in the entire cluster run
                        block = source.read(bytes_per_cluster *
map_entry.run_length)

                        # check to see if the file has any data already
read
                        # non-fragmented files fall under this category
                        if mft_record_num not in files and
map_entry.last_run:
```

```
unprocessedFileQueue.put(UnprocessedFileData(map_entry.mft_record, block))
                        else:
                            if mft_record_num not in files:
                                files[mft_record_num] =
bytearray(map_entry.mft_record.data_attribute().allocated_size())
                            block_offset_start = map_entry.file_offset *
bytes_per_cluster
                            block_offset_end = map_entry.file_offset *
bytes_per_cluster + map_entry.run_length * bytes_per_cluster

files[mft_record_num][block_offset_start:block_offset_end] = block

                            if map_entry.last_run:

unprocessedFileQueue.put(UnprocessedFileData(map_entry.mft_record,
files[mft_record_num]))

                            curr_sector += (map_entry.run_length - 1) *
sectors_per_cluster
                            clusterNum += map_entry.run_length
                            pbar.update(curr_sector)

                    # otherwise, read the cluster and move on
                    else:
                        block = source.read(bytes_per_cluster)
                        curr_sector += sectors_per_cluster
                        clusterNum += 1
                        pbar.update(curr_sector)

                    md5hash.update(block)
                    dest.write(block)
        imaging_end = time.time()
        pbar.finish()
        dest.close()
        source.close()
        print ("Imaging complete. Time taken: {}
seconds".format(imaging_end - start))
        print ("Items remaining in unprocessed queue:
{}".format(unprocessedFileQueue.qsize()))
        print ("Items remaining in processed queue for CASE output:
{}".format(processedFileQueue.qsize()))
        print ("Source hash: {}".format(md5hash.hexdigest()))
        print ("Computing Destination Hash")


        destmd5hash = hashlib.md5()

        dest_numbytes = 0

        with open(sys.argv[2], "rb") as dest:
            print ("Dest file {} open for computing hash".format(destpath))
            dest_numsectors = os.path.getsize(sys.argv[2])/512 + 1
            curr_sector = 0
```

```
            pbar = ProgressBar(widgets=[
                "Sectors Read: ", Percentage(),
                ' ', Bar(),
                ' ', AdaptiveETA(),
            ], maxval=dest_numsectors).start()
            block = dest.read(4096)
            curr_sector += 1
            pbar.update(curr_sector)
            destmd5hash.update(block)
            while block:
                block = dest.read(4096)
                curr_sector += 1
                destmd5hash.update(block)
                pbar.update(curr_sector)
            pbar.finish()
        dest.close()

        print ("Verification complete. Destination hash:
{}".format(destmd5hash.hexdigest()))
        if md5hash.hexdigest() == destmd5hash.hexdigest():
            print("Verification successful, hashes match")
        else:
            print ("Verification unsuccessful.")

        print ("Items remaining in unprocessed queue:
{}".format(unprocessedFileQueue.qsize()))
        print ("Items remaining in processed queue for CASE output:
{}".format(processedFileQueue.qsize()))
        unprocessedFileQueue.join()
        processedFileQueue.join()
        print ("All file processing complete")

        # writing the Case document output

        case_output.serialize(format='json-ld', destination=mpath)

        print ("SPARTA complete. Total time taken: {}
seconds".format(time.time() - start))


if __name__ == "__main__":
    main()
```

APPENDIX II – TESTING RESULTS

All testing results, as well as electronic copies of all source code, are available on the GitHub repository:

https://github.com/jgreenfield11/SPARTA

BIBLIOGRAPHY

[1] E. Zimmerman, "Imager Comparison Testing," [Online]. Available: https://docs.google.com/spreadsheets/d/1wXX5zYql7KIPgrsDdt6S5bTuGt_WRjWaBde1D0fhG5k/edit?type=view&gid=0&f=true&sortcolid=11&sortasc=true&rowsperpage=250#gid=0. [Accessed 31 October 2019].

[2] N. C. S. R. Center, "Glossary Entry for Digital Forensics," 6 March 2019. [Online]. Available: https://csrc.nist.gov/glossary/term/digital-forensics.

[3] S. Garfinkel, "Digital forensics research: The next 10 years," *Digital Investigation*, vol. 7, no. Supplement, pp. S64-S73, 2010.

[4] "WD Black 6TB HDD Review," 25 September 2015. [Online]. Available: https://www.storagereview.com/wd_black_6tb_hdd_review.

[5] "Seagate IronWolf 14TB 7200 RPM 256 MB Cache SATA 6.0Gb/s 3.5" Internal Hard Drive ST4000VN0008," 27 12 2018. [Online]. Available: https://www.newegg.com/Product/Product.aspx?Item=N82E16822184759.

[6] "Tableau Catalog," Guidance Software, [Online]. Available: https://www.guidancesoftware.com/tableau/hardware. [Accessed 09 April 2019].

[7] "Falcon-NEO Forensic Imager," Logicube, [Online]. Available: https://www.logicube.com/shop/forensic-falcon-neo/?v=7516fd43adaa. [Accessed 09 April 2019].

[8] "SuperImager Plus Forensic Units," MediaClone, [Online]. Available: https://www.media-clone.net/Digital-Forensics-Imaging-Investigation-Platforms/1477.htm. [Accessed 09 April 2019].

[9] "Dd," Forensics Wiki, 16 December 2013. [Online]. Available: https://forensicswiki.org/wiki/Dd. [Accessed 09 April 2019].

[10] "dcfldd - Latest version 1.3.4-1," sourceforge.net, 19 December 2006. [Online]. Available: http://dcfldd.sourceforge.net/. [Accessed 09 April 2019].

[11] "dc3dd," Sourceforge.net, 11 July 2018. [Online]. Available: https://sourceforge.net/projects/dc3dd/. [Accessed 09 April 2019].

[12] M. K. Rogers, J. Goldman, R. Mislan, T. Wedge and S. Debrota, "Computer Forensics Field Triage Process Model," *Journal of Digital Forensics, Security and Law*, vol. 1, no. 2, 2006.

[13] "Evimetry," 27 December 2018. [Online]. Available: https://evimetry.com/how-evimetry-works/.

[14] J. Grier and G. G. Richard III, "Rapid forensic imaging of large disks with sifting collectors," *Digital Investigation*, vol. 14, no. Supplement 1, pp. S34-S44, 2015.

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

[16] L. Tobin and P. Gladyshev, "Open Forensic Devices," *Journal of Digital Forensics, Security and Law*, vol. 10, no. 4, 2015.

[17] W. Ballenthin, "williballenthin/INDXParse," Github, 1 October 2018. [Online]. Available: https://github.com/williballenthin/INDXParse. [Accessed 09 April 2019].

[18] B. Carrier, "Open Source Digital Forensics," Brian Carrier, [Online]. Available: https://www.sleuthkit.org/. [Accessed 09 April 2019].

[19] E. Casey, S. Barnum, R. Griffith, J. Snyder, H. van Beek and A. Nelson, "Advanced coordinated cyber-investigations and tool interoperability using a community developed specification language," *Digital Investigation,* pp. 14-45, 2017.

[20] S. Garfinkel, "Digital forensics XML and the DFXML toolset," *Digital Investigation,* vol. 8, no. 3-4, pp. 161-174, 2012.

[21] Z. Syed, A. Padia, T. Finin, L. Mathews and A. Joshi, "UCO: A Unified Cybersecurity Ontology," in *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[22] Garfinkel, Farrell, Roussev and Dinolt, "Bringing Science to Digital Forensics with Standardized Forensic Corpora," *Digital Forensics Research Workshop,* 2009.

[23] "PerfectDisk Tools," Raxco Software, [Online]. Available: https://www.raxco.com/tools-utilities. [Accessed 09 April 2019].

[24] "Microsoft Windows Dev Center," 17 08 2019. [Online]. Available: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines.

[25] "Tools & Utilities," Raxco Software, [Online]. Available: https://www.raxco.com/tools-utilities. [Accessed 17 08 2019].

[26] "Defraggler," Piriform, [Online]. Available: https://www.ccleaner.com/defraggler. [Accessed 17 08 2019].