CREATING HIGHLY SPECIALIZED FRAGMENTED FILE SYSTEM DATA SETS FOR FORENSIC RESEARCH

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies at

University of Idaho

by

Douglas Drobny

May 2014

Major Professor: **Jim Alves-Foss**, Ph.D.

AUTHORIZATION TO SUBMIT THESIS

This thesis of Douglas Drobny, submitted for the degree of Master of Science with a Major in Computer Science and titled "Creating Highly Specialized Fragmented File System Data Sets for Forensic Research", has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor	______________________ Dr. Jim Alves-Foss	Date ________
Committee members	______________________ Dr. Paul Oman	Date ________
	______________________ Dr. Marty Ytreberg	Date ________
Computer Science Department Administrator	______________________ Dr. Gregory Donohoe	Date ________
Discipline's College Dean, College of Engineering	______________________ Dr. Larry Stauffer	Date ________

Final Approval and Acceptance by the College of Graduate Studies

	______________________ Dr. Jie Chen	Date ________

ABSTRACT

File forensic tools examine the contents of a system's disk storage to analyze files, detect infections, examine account usages and extract information that the system's operating system cannot or does not provide. In cases where the file system is not available, or information is believed to be outside of the file system, a file carver can be used to extract files. File carving is the process of extracting information from an entire disk without metadata. This thesis looks at the effects of file fragmentation on forensic file carvers. Since current fragmented data sets have limited availability and are time consuming to create, a tool for creating fragmented file system data sets is introduced. This thesis describes that tool, Whetstone, which is intended to make data sets for forensic research easier to create and reproduce. Whetstone aims to simplify the process for creating a test data set that includes file fragmentation or has a specific layout. Easy to use file manipulation controls and reproducible data sets are the key components of Whetstone and are available through a user friendly graphical interface. Types of fragmentation and the effects they have on file carvers are also discussed. Whetstone is then used to create data sets with various types of fragmentation in order to demonstrate the effects of fragmentation on file carver performance.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. Jim Alves-Foss, for his support, encouragement, and patient guidance throughout my graduate studies.

I would also like to thank my other committee members for their valuable comments on my thesis.

I would like to thank all my instructors for providing me knowledge about Computer Science and building my research skills.

Thank you to the students in the Scholarship for Service program for keeping me focused.

Last, but certainly not least, I would like to thank parents and families for their understanding, support, encouragement, and love which have helped me make this thesis a reality.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

With the rise of computer related crimes, the need for effective electronic evidence gathering tools has increased. In general, forensic tools are used to gather and examine information about the past from various sources. Forensic science covers many areas including forensic accounting, anthropology and toxicology. Computer forensics tools aim to retrieve evidence and information (such as images, emails and web history) from machines involved in cybercrime and machines encountered during a non-electronic investigation. In addition to law-enforcement applications, computer forensic tools are used to protect information, understand computer usage and help recover systems from accidents and misuse. Companies are using forensic toolkits to understand the extent of cyberattacks, recover deleted files, track data, detect malware and more. The versatility of forensic tools allow for plenty of areas of improvement and study.

This thesis focuses on the file forensics branch of host-based computer forensics. The two categories of computer forensics are network and host-based forensics. Network forensic tools operate on network traffic to detect attacks, analyze data entering/exiting a network, and to gain further understanding of what is happening on the network. These tools can operate in real time or off of stored network data. A common corporate application of network forensic tools is to determine what kind of information left the internal network after an attack or information leak. Host-based forensic tools use information found on the host machine to extract information of how that machine has been operating. File forensics is a subset of host-based forensics and examines the contents of a system's disk storage to analyze files, detect infections, examine account usages and extract information that the system's operating system cannot or does not provide. File forensic toolkits often traverse a file system for an investigator in a consistent and easily searchable manner. For the purpose of this thesis when discussing forensics, we will limit ourselves to file forensics; system logs and network forensics have a different data set and suffer from different problems than file forensics and won't be discussed further.

The rapid increase in the size of hard disks has increased the amount of data that forensic tools are having to search and recover. Faster forensic tools means less time recovering data and more time for investigators to examine recovered information. File forensic tools are often used to obtain legal evidence or information that may become legal evidence. Even forensic analysis, without the intention of obtaining legal evidence, may lead to finding information that may eventually be used in a legal manner. This applies pressure on forensic tools for additional accuracy, improved preservation of evidence and a simpler presentation of evidence. Since data on different systems can be so diverse, a file forensic tool that correctly retrieves all of the information from one system may not retrieve all of the information from another system. This, combined with the tool's goal of extracting as much information as possible from a system, makes for constant areas of improvement for new tools. Forensic tools are also dealing with a constantly changing environment, with new file systems and file types being created.

Retrieving all of the evidence from a system is only part of the responsibility of a forensic tool. Forensic tools need to maintain the data integrity of the data evidence. This causes forensic tools to constantly examine the impact of the tool on data and prove that the evidence was not modified [26]. The courtroom setting also imposes another challenge for forensic tools; evidence needs to be presented in a manner that is easy for officials or juries to understand [26]. Complex algorithms and file system tricks (like modified or inconsistent timestamps) can cloud understanding and reduce the usefulness of the evidence.

Whether it is a suspected criminal's hard drive or a flash drive dropped in a puddle on the way to work, sometimes retrieving files from a device is not as simple as plugging it in. Some data acquisition methods use the operating system and file system to help retrieve data. In cases where the file system is not available or information is believed to be outside of the file system, a file carver can be used to extract files. File carving is the process of extracting information from an entire disk without requiring support from operating system metadata. In addition to normal files, file carvers can find deleted, overwritten and corrupted files using knowledge of file structures, statistics and graph theory.

File fragmentation occurs when the sections of disk that a file occupies are not contiguous. While fragmentation can be reduced, it is still a possibility when recovering data from a file system, so file carvers need to be able to extract fragmented information. A user may also use file fragmentation as an anti-forensic method to slow down the carving process.

The result of carving a fragmented file is tested for correctness by forensic validation algorithms. Validation algorithms aim to automate the process of verifying if a file is recovered correctly. Manual validation involves an investigator opening a file and looking to see if it appears to have been correctly recovered. Fast automated validation techniques can be used during the file carving process to quickly determine if a fragment was correctly allocated to a file and are an important part of the file carving process.

This thesis examines the problems facing current file forensic algorithms and explores potential research areas. The goal of this thesis is to examine file carving algorithms and demonstrate a need for file carvers that can handle various types of file fragmentation. A tool for creating specific data sets to test file carving algorithms on is also introduced. Since forensic tools work with files that can be in various conditions (incidental fragmentation, anti-forensic fragmentation, corrupted, etc.) researchers working on algorithm advancements need to have test data sets that represent these possible conditions. The tool will give researchers an alternative to the few available forensic data sets online with fragmented files. The custom data sets created by the tool give researchers a viable option for making test cases with fragmented files.

1.1 Thesis Overview

The remainder of this thesis is organized as follows. Chapter 2 gives background information on file systems and file carving solutions. Chapter 3 details the criteria for a research oriented data set creation tool. Chapter 4 is a description of the tool and its usage. Chapter 5 discusses validation testing for the tool and the results. Chapter 6 summarizes the concepts discussed in this thesis, the tool presented and possible future additions to the tool.

CHAPTER 2
BACKGROUND

2.1 File System Overview

A file system is the process and data structures responsible for the management of files on a disk. New file systems are constantly being developed and can be very different from current popular file systems. Since this thesis focuses on the use of file carvers, which ignore the file system, this thesis uses a simple file system with a file allocation table to explain concepts. While not all file systems use a file allocation table, most use a similar feature to track files. This chapter gives a basic overview of how files are stored on a file system and problems a file system encounters relevant to file carving.

2.1.1 How files are stored

A file system typically uses clusters on the hard disk for file allocation. Small sections of disk surface area known as track sectors are grouped together to form clusters of a certain size when a file system is installed. Figure 2.1 shows how different sections of a hard disk platter are named. A track is one of the circular rings on the hard drive and a physical sector is section of the disk, similar to a slice of pie. These divisions let the file system manage of the location of data on the hard drive. The location where a specific track and single physical sector overlap is a track sector and groups of track sectors are clusters.

Figure 2.1 shows a cluster made up of three sectors. For the purposes of this thesis sectors refer to track sectors and not physical sectors. Clusters serve as the smallest unit of space a file can occupy on the system. When a file is created, the file system determines the number of clusters required to store it. This is often more space than the file originally occupies, in order to allow for the file to be expanded. Then the file system chooses an appropriate set of file clusters to allocate to the file. Most files are assigned to a set of contiguous clusters, but it is possible to for the file system to split the file into more than one set of contiguous clusters. Splitting of a file's clusters is called fragmentation, and is discussed later in this chapter.

The file system keeps track of which clusters are being used and by which files. The file allocation table (file table) contains the information that links files to their clusters. Figure 2.2 shows a single entry in the equivalent of a file allocation table (668333). The metadata shows that the file is 403 bytes and is allocated in the data block 127754. Additional information

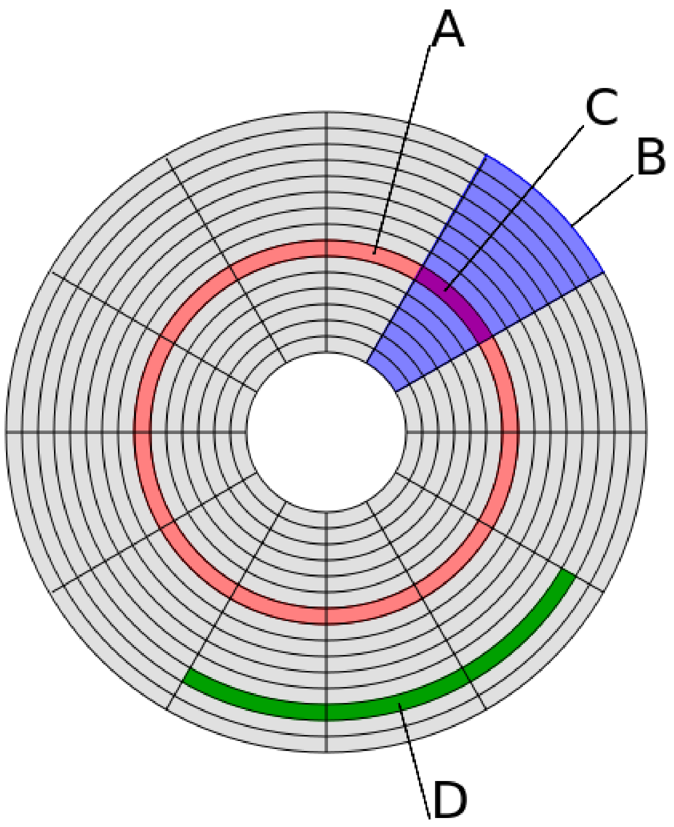
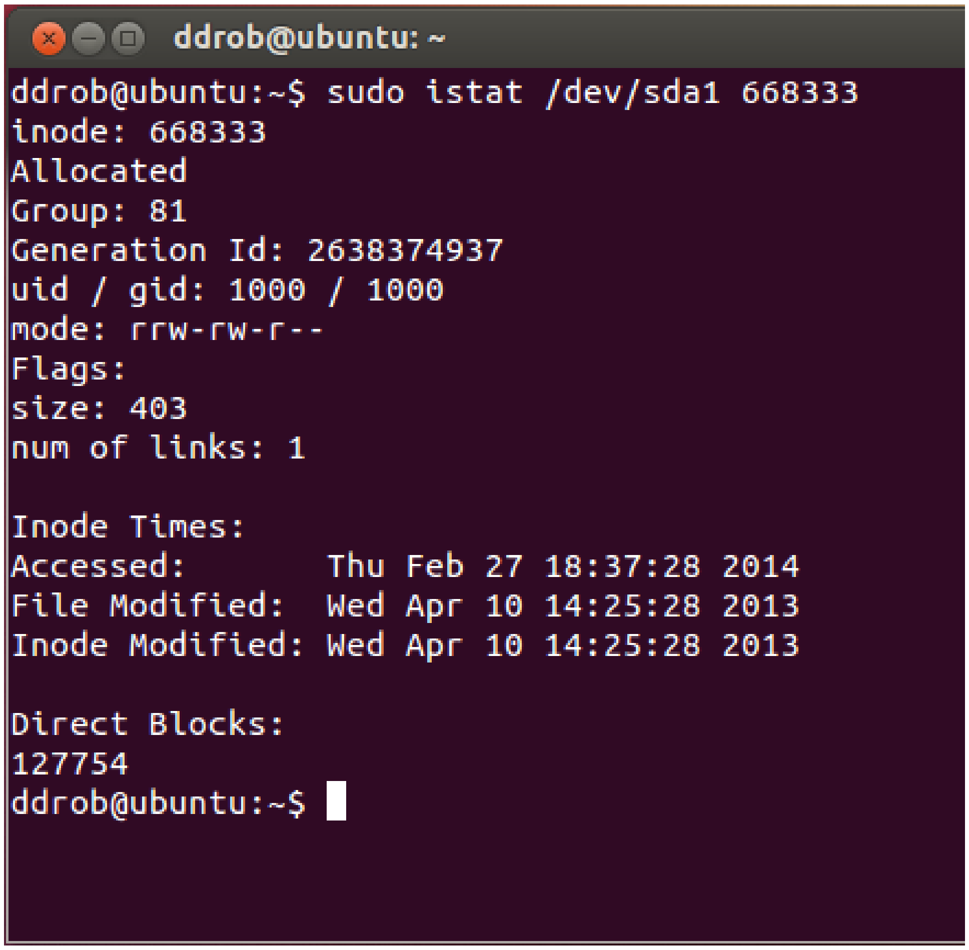


Figure 2.1: Disk structure labeling. A: Track. B: Physical Sector. C: Track Sector. D: Cluster. [18]

about the file system is found in the file system metadata. While the file system metadata is file system specific, typical file system metadata includes the location of the root node, logging files, bad clusters and available clusters. When a file is created, metadata is created that contains information about the new file such as creation time, author, and file extension. Some file systems store file metadata separate from the file data, others surround the file data with the metadata and place it in the same clusters. In addition to hard disks, non-traditional memory devices like USB flash drives or solid state drives have file systems that use virtualized sectors/cluster that behave in the same way.



```
ddrob@ubuntu:~$ sudo istat /dev/sda1 668333
inode: 668333
Allocated
Group: 81
Generation Id: 2638374937
uid / gid: 1000 / 1000
mode: rrw-rw-r--
Flags:
size: 403
num of links: 1

Inode Times:
Accessed:       Thu Feb 27 18:37:28 2014
File Modified:  Wed Apr 10 14:25:28 2013
Inode Modified: Wed Apr 10 14:25:28 2013

Direct Blocks:
127754
ddrob@ubuntu:~$
```

Figure 2.2: Formatted file system metadata for test.cpp

2.1.2 File management problems

A file system that only creates files has very few problems managing disk space. Difficulties arise from changing file sizes and deleting files. When information is added to the end of a file, the file system needs to store the additional data somewhere. To allow for a file to expand and contract over time the file system will allocate additional space to each file upon creation. Since the smallest unit a file system can allocate is a cluster, if a file only uses one fourth of the space in its last cluster, the remaining space can be used for expansion. The area between the end of a file and the end of its final cluster is referred to as slack space.

For example, assume we have a file system using a cluster size of 256 kilobytes and a file that is 650 kilobytes. The file system would allocate three clusters to the file (which would total 768 kilobytes of space). The leftover 118 kilobytes are the slack space and will be used if the file is expanded.

Deleting files can also cause problems for a file system. When a file is deleted, the information in its clusters is not usually deleted, instead its clusters are just released into the pool of available clusters. The original data still resides in those clusters (this is why it is possible to undelete files). When a new file is assigned clusters, if it is given the clusters of a deleted file, the new file simply overwrites the old data. This can leave some of the old file's information in the new file's slack space. Deleting files can cause another problem. Since file systems typically allocate the first set of clusters that will hold the file, it is probable that the clusters directly after a deleted file contain another file. When the file system begins to fill, the system may need to use the newly deleted clusters. If the new file is larger than the hole left by deleting files then some of the new file's clusters will not be contiguous. This process, a type of disk fragmentation, generates incomplete sections of files called fragments.

2.1.3 Example of File Fragmentation from File Management

An example of fragmentation (see Figure 2.3) is: File A is a 3 cluster file that is surrounded by files 1 and 2. When file A is deleted 3 clusters are released. File B is then created and requires 5 clusters worth of space. Assume that file B uses the three clusters File A occupied, then the final two clusters of file B must be placed in the next set of available clusters, which are not next to the File A clusters. If file B does not use the 3 clusters from file A then the 3 cluster hole must be tracked and a method for determining when to fill the hole must be created. The effect of file B filling in file A's clusters is depicted in Figure 2.3. Different file systems have different methods of filling the holes left by removing files. One method for filling holes would be to instantly fill holes when a file is created, while another method may be to only fill holes if a file would be placed in at most two separate holes.

2.2 File Carving

This section introduces file carving concepts, file carver techniques, and gives a few example file carvers.

2.2.1 Overview

Early data recovery tools attempted to restore the file system on a drive to allow the user to read the contents, but the complexity of modern operating systems makes these tools unusuable

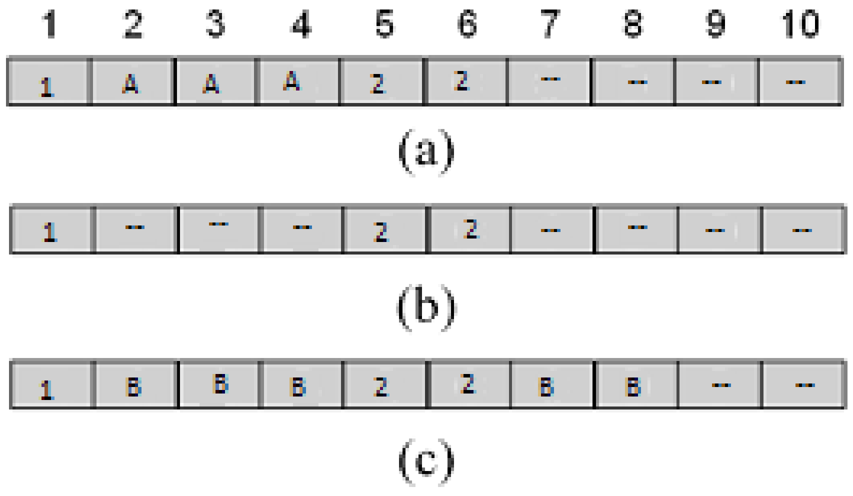


Figure 2.3: (a) Files 1,A,2 on disk. (b) File A is deleted. (c) File B is added

without risking modification to the data [15]. With file systems that use a table showing where the contents of a file are kept, if that table is destroyed, then the contents of a file are not directly recoverable. To recover data in these situations forensic tools were developed that rely on the file content and structure instead of operating system metadata. These tools, called file carvers, are also capable of examining unused sections of a drive for files. File carvers can be used on an entire disk, or just on specific sections of a disk to increase the speed of carving. Similar to other forensic tools, many file carvers have a known file filter that will recognize common files (such as Window's system files) and quickly carve them.

2.2.2 File carving techniques

A basic carving technique is to detect files based on header and footer byte sequences. File carvers match rules to find byte sequences specific to a type of file to determine the start and end of a file. A file carver that detects a header with file size information would carve a file out of that header and the space following it. If both a header and footer sequence are detected, the file carver will create a file out of the data in between the header and footer. This technique can be visualized by thinking about finding a report in a stack of papers. The file carver would search through the stack of papers and find the first and last pages of the report. The file carver would take out the section of papers between the first and last pages and decide that was the entire report. This type of carving assumes that the file is not fragmented and the information between the header and footer is not destroyed [17].

Header and footer byte sequence analysis is considered a classification-based file carving technique. Classification techniques attempt to determine the type of file each fragment belongs to; they are further discussed in Section 2.4.1. The other techniques are identification and validation. Identification techniques aim to identify the specific file that a fragment belongs to and are discussed in Section 2.4.2. Validation carvers are used in conjunction with another carving technique and analyze the results of the other carver to determine if it was a likely successful file recovery; more information on validation carvers can be found in Section 2.4.3.

2.2.3 File Carving Tools

Scalpel and Foremost are two of the most common file carvers. Foremost was developed by the United States Air Force Office of Special Investigations and the Center for Information Systems Security Studies and Research in 2001 [9]. Scalpel was released as an optimized version of Foremost, with improved performance and memory usage [23] [17].

Forensic toolkits often include file carvers to increase the information retrieved. Since a basic file carver like Scalpel will result in a large amount of incorrect files in a realistic analysis (fragmented, corrupted, etc.) commercial forensic tools use more advanced carvers. Encase and FTK are toolkits that use advanced carving techniques to handle disk fragmentation and implement filters for known files [17]. To better understand file carving techniques, we first provide a more detailed discussion of file fragmentation.

2.3 File Fragmentation

The following sections give an overview of data fragmentation, the causes of fragmentation, and how file carvers are affected by data fragmentation.

2.3.1 Fragmentation Overview

There are three types of data fragments: deleted fragments, file fragments, and micro-fragments. Deleted fragments are created when a file is deleted from a file system. During normal file deletion, the file system makes the location on the disk available for file allocation but does not erase the contents of the file. This process is what allows for file 'undeletion'. When clusters of the deleted file are reallocated, some clusters will remain unused and still contain portions of the original data. These clusters of deleted data are called deleted fragments.

File fragments are created when the file system has to split the file into non-contiguous clusters on the disk. Although a file system is not affected by file fragmentation, a file carver does not know where the other file fragments are and can run into difficulties in reassembly. As a disk is filled and files are created and deleted, the number of file fragments is increased. A disk defragmenter utility is designed to reorder the files of the current disk to reduce the number of file fragments. Some operating systems and optimization tools regularly defragment storage devices, while other times defragmentation is left to the user to manage.

Micro-fragments are similar to deleted fragments in that they are remnants of previous files. Where deleted fragments are composed of the information found within a deleted cluster, micro-fragments are found in the slack space after a deleted cluster has been allocated to a new file. If the new file writes over the deleted file's contents and does not fill the entire cluster, which is common in the last cluster of a file, the remaining bits in the cluster are considered a micro-fragment [10].

2.3.2 Impact on carvers

Fragments can destroy the effectiveness of basic file carvers. Fragmentation causes three types of problems for file carvers that rely on header and footer information and multiple types of fragmentation can increase the chance of fragmentation problems for a file carver. The first problem is that additional headers and footers can be introduced in the file system through deleted fragments and micro-fragments. Deleted fragments and micro-fragments still retain data from before they were deleted or overwritten. A file carver, which is unware the fragments were deleted, treats these deleted fragments as originals. If the data in those deleted fragments contains header and footer information the file carver will add these fragments as additional header and footer fragments. These additions can cause a file carver to incorrectly determine when a file starts or finishes. For example, if a file carver correctly detects the start of a JPG file and a micro-fragment containing characters associated with a JPG footer is found before the correct JPG footer, the file carver will use the micro-fragment as the footer and not the actual JPG file footer fragment.

Another problem fragmentation can cause is when a file is surrounded by another file's fragments. Consider a situation where file A is split into two fragments and file B is a contiguous

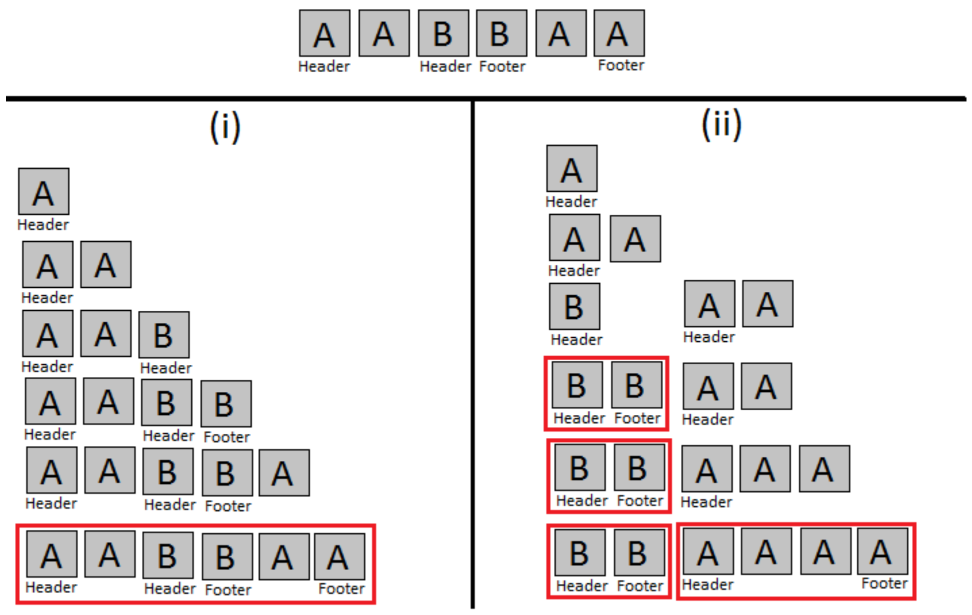


Figure 2.4: (i) File carver does not detect headers while matching a header. (ii) File carver detects multiple headers.

file (see Figure 2.4). If the fragments of file A surround file B, a file carver may have difficulty detecting the existence of file B. If file A is a JPG and file B is a PNG the file carver will detect a JPG header and begin searching for a JPG footer. Since the JPG footer is found after the entirety of file B, the contents of file B would be included inside of the carved file, which is incorrect. Figure 2.4 column (i) shows a carver that matches a header to a footer without searching for new headers, and column (ii) shows a carver that continues to search for header and footer information when matching a header to detected embedded or surrounded files.

The third problem for file carvers that rely on header and footer information is that fragments that do not contain header and footer information are difficult to place. These fragments can be included within the data space for carved files causing their additional data to corrupt the resulting file. If these fragments are found outside of other files a basic file carver will not attempt to carve these fragments into a file but will throw the information out. For a data set with fragments without header and footer information, a more advanced file carving technique needs to be used. These advanced techniques are described in more detail in Section 2.4.

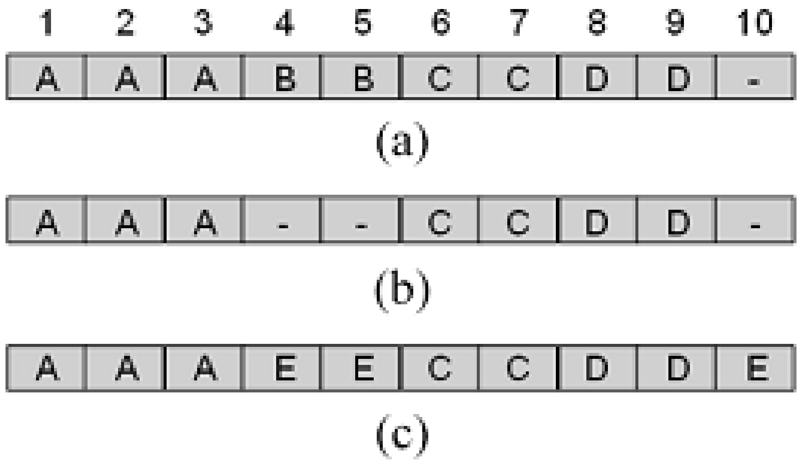


Figure 2.5: (a) Files A,B,C,D on disk. (b) File B is deleted. (c) File E is added and is bi-fragmented

2.3.3 Types of fragmentation

Files are divided into three categories of fragmentation: Contiguous, bi-fragment, and multi-fragment. Contiguous files are files where the fragments appear in the correct order and do not contain any fragments of other files between the first and final fragments. Bi-fragmentation is the most common type of fragmentation where two sections of the file are separated by fragments of other files. Bi-fragmented files are split in two by gaps filled with fragments of other files (or entire files). Bi-fragmentation is illustrated in Figure 2.5(c) for File E. A multi-fragmented file has several sections of fragments separated from other sections. Multi-fragmented files are most common on devices that are near full capacity. Bi-fragmentation and multi-fragmentation describe the number of noncontiguous fragments a file is divided into and fragments are placed into these categories regardless of the ordering of the noncontiguous fragments.

2.4 Current Methods

As mentioned earlier, there are three general categories of file carving methods used: classification, identification and validation. These methods are described in this section.

Table 2.1: Foremost's magic numbers by file extension.

Ext	Max Size	Header	Footer
jpg	200000000	\xff d8 ff e0 00 10	\xff d9
gif	5000000	\x47 49 46 38 37 61	\x00 3b
gif	5000000	\x47 49 46 38 39 61	\x00 3b
htm	50000	`<html`	`</html>`
pdf	5000000	%PDF	%EOF\x0d
pdf	5000000	%PDF	%EOF\x0a

2.4.1 Classification Methods

Classification is the process of determining what type of file a fragment belongs to. Magic number, byte frequency distribution and metric-based classifications are common approaches to file classification.

The most popular method of fragment classification is the magic number approach. This approach involves a file carver looking for specific groups of bytes to identify a file type. The groups of bytes are referred to as magic numbers. UNIX systems may store magic numbers in /etc/magic or /usr/share/misc/magic. These magic numbers are designed to be detected in the first 16 bytes of a file [19]. The magic numbers for a file type are typically related to header and footer information, but some are related to encodings and structures within a file. The file carver Foremost uses a configuration file filled with patterns for detecting magic numbers. For example the Foremost configuration entry for jpg files is 'jpg y 200,000,000 \xff\xd8\xff\xe0\x00\x10\xff\xd9' [3]. This line states that when looking for a jpg file it checks for the hex string '\xff d8 ff e0 00 10' to start the file and the hex string '\xff d9' to end the file. The configuration line contains a lowercase y after the extension which indicates that the magic numbers are case sensitive. The next field contains a number that determines the carved file's maximum file size, which in this example is 200,000,000 bytes. Figure 2.1 shows the magic numbers for several popular file extensions. This method is accurate at determining file types but only works when a fragment contains the header and footer magic numbers. PhotoRec by CGSecurity also uses magic numbers to determine where files start and stop. PhotoRec also uses header and footer information to determine the cluster size of a drive which it then uses to carve files on a cluster by cluster basis [6]. Different than Foremost, PhotoRec will retain information about fragments previously searched and use these when encountering new fragments. This allows PhotoRec to handle some types of fragmentation.

TrID identifies file types based off of a large database of known file types [22]. Users send in samples of known file types and TrID creates signatures for that file type based on the information received. The current database has 5,276 identifiable file types. Essentially TrID uses a database of file signatures instead of the /etc/magic and configuration files Scalpel uses. Users can even identify a file's type using TrID online. TrID operates on entire files and not on fragments alone, but the concepts behind TrID can be applied to identifying fragments.

Advanced classification techniques have been tested on fragments based on statistics and machine learning. These techniques require a training set for the algorithm to learn from and files need reliable patterns for the algorithms to detect. The training set is a set of known data that the algorithm runs on and the algorithm tweaks values and approximations until it gets the correct results. Then the algorithm uses those values and approximations (which are known to produce correct answers) on the actual data set.

Two statistical approaches to fragment classification are byte frequency distribution and metric-based. Byte frequency distribution techniques look at the rate in which certain byte patterns appear. Unlike the magic number, byte frequency distributions are not told the patterns to search for but instead learn them through a training set. Karresand and Shahmehri introduced two methods based on byte frequency distribution [12] [11]. Their first method, Oscar, compares the distribution of the values found within a fragment. Essentially histograms are created from fragments whose types are known (training set) and then a histogram of the unknown fragment is created. The trained histogram that most closely matches the fragment's histogram classifies the fragment as the training histogram's type [12]. Their second method uses the rate of change between bytes instead of the actual byte frequencies. In this method the histograms are created using the difference in values between sequential bytes [11]. This allows the algorithm to detect patterns in byte ordering that the Oscar method does not. In their testing they found that Oscar produced a detection rate of 87.3% and false positive rate of 22.1% while the rate of change method had a 92.1% detection rate and 20.6% false positive rate [11]. These test results were performed primarily on JPEG, ZIP and windows executable files. The majority of these algorithms success comes from being able to learn and identify header and footer magic numbers.

McDaniel et al. used a byte frequency detection algorithm with cross-correlation analysis to classify fragments [14]. This method takes into account the relationship between two bytes (such as the characters '<' and '>' in HTML documents) to improve effectiveness. An array of correlation values representing each relationship is created and used when determining how close two centroids are. McDaniel's cross-correlation technique was slower than traditional byte frequency analysis but increased success. Tests without the cross-correlation technique were 27.5% successful and 45.83% with cross correlations [14]. Despite using a more complicated method, the results of byte frequency distribution are not significantly more effective than the magic number approach [24]. Researchers have been integrating byte frequency distribution techniques into larger algorithms [4] [29]. Amirani et al. proposed a new technique that combines byte frequency distribution, feature extraction and neural networks to classify files [4]. They combined byte frequencies and features created by neural networks to create classification centroids. In their experiments the new technique produced a 98.33% detection rate and is fast enough to be used in a realistic scenario, but was tested on only a few file extensions.

Metric-based approaches use statistics to attempt to classify file types. By taking statistical measurements from training files and comparing the results to fragments, metric-based approaches are able to determine some qualities of the fragment's file type. Moody and Erbacher identified average, kurtosis (peakedness), distribution of averages, standard deviation, distribution of standard deviations, and byte distribution as the most influential statistical values [19]. Metric-based approaches work well on file types with a frequent repetition in data. In a test of CSV, HTML, and TXT files, Moody and Erbacher were able to correctly classify 96% of CSVs, 84% of HTMLs, and 80% of TXTs. These methods are able to classify general information such as if a file is text, binary, or an image, but can have difficulty narrowing down specific file types [24].

Sportiello and Zanero implemented a machine learning classifier based on support vector machines [27]. A support vector machine is a mathematical function that determines which of two categories an object can be classified as [30]. Their classifier uses several support vector machines that classify a block of data based on byte frequency distribution, rate of change, mean byte value, entropy, complexity and a few other qualities. Each file type has a support vector machine that classifies if a data block (fragment) is of that type or not of that type. The classifier introduced produced an average of 90% true positives with 12.4% false positives [27]. They used BMP, DOC, EXE, GIF, JPG, MP3, ODT, PDF and PPT file types.

Table 2.2 contains the results of tests done by various fragment classification techniques discussed in this section. It is important to keep in mind that these methods were not tested by the same group or on the same data set.

Table 2.2: Number of files recovered by extension

Author	Method	Success Rate	Notes
McDaniel et al. [14]	Byte Frequency Distribution	27.5%	
McDaniel et al. [14]	BFD w/Cross-Correlation	45.83%	
Amirani et al. [4]	BFD w/Neural Network	98.3%	Limited File Types
Karresand and Shahmehri [12] [11]	BFD-Oscar	87.3%	JPG,ZIP,EXE only
Karresand and Shahmehri [12] [11]	BFD-Rate of Change	92.1%	JPG,ZIP,EXE only
Moody and Erbacher [19]	Metric Based	96%	CSV only
Moody and Erbacher [19]	Metric Based	84%	HTML only
Moody and Erbacher [19]	Metric Based	80%	EXE only
Sportiello and Zanero [27]	Support Vector Machine	90%	

2.4.2 Identification Methods

File identification is the process of determining which specific file a fragment belongs to. This process can be performed before or after classification. Classifying fragments can reduce the combination of fragments available for identification (since JPG fragments will match up with JPG files). Identification can increase the success of classification (since the classifier has a larger 'fragment' to work with). The three types of identification approaches are: file structure, graph theory, and bi-fragmentation.

File structure identification is the type of identification performed by the basic file carvers discussed earlier. A header and footer are found by the file carver, then all fragments between the header and footer are identified as belonging to that file. While simplistic, it is very quick and effective on disks with little fragmentation.

Identification methods based on graph theory have to carefully balance the accuracy of the method with combinatorial complexity. The goal of a graph theory identifier is to translate the fragments and their relationships into graph vertexes and edges. An effective carver must be able to scale to large drives and graph theory methods are at risk of having a very large number of possible combinations on large drives. Then the identifier uses an established graph algorithm to determine the optimal solution, which is then used as the correct identification for each fragment [17].

Shanmugasundaram and Memon relate the identification of file fragments to a Hamiltonian path problem [25]. A Hamiltonian path is a path through a graph that visits each vertex once and only once. In this algorithm the fragments are mapped to the vertices of the graph and the edges of the graph are weights based on the likelihood of fragments being connected. These weights are based on prediction models and different models will result in different solutions. Once the graph is constructed the algorithm attempts to determine the maximum weight Hamiltonian path. This would mean each fragment gets used exactly once. A maximum weight Hamiltonian path is the path that has the largest sum of the edge weights while visiting each vertex once. This would be the solution with the highest likelihood of fragments being related, while only using each fragment once.

The Hamiltonian path problem is more effective when determining the ordering of fragments belonging to a small number of files. With many files the problem is more directly related to a k-vertex disjoint path problem. The k-vertex disjoint path problem takes a graph and essentially finds a number of separate Hamiltonian paths that end up using each vertex. Identification techniques using this method determine the number of Hamiltonian paths to find, based on the header information in the files. In Memon and Pal's paper on fragmented image reassembly, they use information from the edge pixels of each fragment to weigh the edges on the graph in a k-vertex disjoint path solution to multiple fragmented images [16].

Thing et al. implement an identification based file carver called "Progressive Joint Carver" [28]. The carver identifies the header fragments in the data set and makes these fragments the base for reconstructed files. Then each fragment has its weight calculated and is identified as belonging to the file with the closest matching header. The weights are calculated based on the last X bytes of one fragment and the first Y bytes of another fragment. Fragments with close weights are attached together to form a joint. The Pro-Joint carver was able to recreate a test image in 1.5 minutes compared to a commercial product recreating the image in 2.8 minutes [28].

According to Garfinkel [9], bi-fragmentation methods are specifically designed for detecting files that have been fragmented into two sections. Bi-fragmentation methods are an expansion of the file structure methodology. Where file structure methods add everything between the header and footer, a bi-fragmentation method assumes that file is fragmented and there are clusters between the two that do not belong. The largest challenge a bi-fragmentation method faces is determining the end of the first fragment and the start of the second fragment. A bi-fragmentation method outlined by Garfinkel is bi-fragment gap carving [9]. Bi-fragment gap carving uses validation techniques to exhaustively search for the fragmentation points. Essentially, the method tests all possible fragmentation points given a certain size gap between the end of the first fragment and start of the second fragment. The size of the gap is increased until a satisfactory file is recovered.

2.4.3 File Validation

File validation is the process of selecting the files chosen by the file carver and verifying that they were correctly recovered. Manual validation includes a forensic investigator opening a recovered file and visually inspecting it for potential errors. Because the process of manual validation is very time consuming, automating file validation greatly increases the speed at which a disk can be analyzed. This section discusses validating with headers and footers, data dependency, container structures, decompression, and semantics.

Using headers and footers to validate files is the fastest and most ineffective way to automate the validation process. This technique checks the header and footer information of the reconstructed file, and if they are compatible it marks the file as validated. Created in 2005 by Richard and Roussev, Scalpel is an expansion on Foremost that uses header and footer validation [23]. This helps Scalpel meet its design requirement of running on low end machines or machines with few resources [23]. Since this type of validation does not look at the data contained between the header and footer information it is ineffective at finding corrupted, missing or additional data within a file [9]. Data dependency validation occurs after header and footer validation, it checks to see if certain parameters associated with a file type are met by the file. In the header information of some file types, like BMP, are fields associated with file length or other internal structure information [5]. Using data dependency validation can reduce the number of false positives generated by header and footer validation.

Container structures are files that often contain files within them. For example zip files contain several compressed files and the information explaining which files were zipped up. Container validation looks at the header and footer information for the container and checks to see if the contents might possibly belong in the container. For example: examining a ZIP file with header information indicating it is 512 kilobytes and the file contains less than 512 kilobytes indicates that the ZIP file is a valid container. While more effective than a header and footer validation technique, the contents of the container need to be validated by another technique, or manual inspection [9].

One of the techniques to further validate container data uses decompression. The general idea is to use a decompressor on the file and see if it returns any errors. This process can be sped up on some compression techniques by analyzing the data for markers that indicate it underwent that specific type of compression (e.g., a JPG compressed file containing the correct Huffman markers). While more time consuming than the other validation techniques this method actually looks at the data within the file and validates that information [9].

Semantic validation is very difficult to implement. Semantic validation is based on the theory that certain assumptions can be made about the documents, based on known characteristics (like language). If a semantic validator finds a fragment that ends with one word (frag) and the next fragment begins with another word (ment) than the validator assumes that the fragments are in the correct order because the two words can be combined into another word (fragment). This type of validation is not widely used [9].

Often multiple types of validation are combined to reduce false positives. In their paper, Chen et al. create a PDF validator that uses multiple types of validation [7]. They use header and footer validation to find the boundaries of the PDF fragments. Data dependency validation is used in conjunction with container validation to determine the internal structure of the PDF. They also use zlib/deflate decompression techniques to confirm whether certain sections of data are correct. Combined they were able to create a PDF validator that successfully validated PDFs that were fragmented up to three times [7].

2.5 Summary

A file system allocates file data to groups of hard disk sectors known as clusters. The clusters are the smallest unit a file system will allocate data to. The file system is in charge of choosing which clusters a file belongs to. A file with data spread over non-contiguous clusters is a fragmented file, with each contiguous group of allocated clusters making up a fragment. File fragmentation typically occurs when a file system allocates extra space for file growth, deletes a file, or allocates space for a new file while running low on space. Some forensic recovery tools that operate without file system metadata, called file carvers, work less effectively when a disk has many fragmented files.

File carvers were designed to recover information where crucial parts of information from the file system are unavailable. These tools are typically used on devices with corrupted data, physical damage or unknown file systems. Foremost and Scalpel, two popular free file carvers, operate based on matching found data with header and footer data associated with known data types [23]. Identification based file carvers try and match each file fragment with files partially recovered by the tool. These carvers use graph theory to optimize which recovered files a fragment should be matched up with. Classification based file carvers focus on determining each fragment's type based on statistics, machine learning or strict file format rules. Statistics and training machine learning algorithms play a key role in the effectiveness of a classification based file carver. Once a file is believed to be recovered it may go through a file validation process which determines if a file was correctly recovered. These validation processes may involve visually inspecting a file, checking a file for proper header and footer information, or running the file through a format specific validation test. Some file carvers are based around recovering as many possible files and using automated validation to narrow the result pool.

All of these types of carvers share a similar problem: file fragmentation greatly reduces their effectiveness. File fragments can alter recovered data, increase false recovery rates and slow carvers down to a crawl. To improve file recovery quality and speed, advancements to file carvers must be made. New techniques and new carvers need to be rigorously tested on varied data sets including sets with different types of file fragmentation. The next chapter describes beneficial characteristics for a tool that creates file carver test data sets. These data sets can be used to test and improve any type of file carver from a multitude of file fragmentation types.

CHAPTER 3
TOOL FUNCTIONALITY GOALS

The goal of this project is to design and create a tool that aides file forensic researchers in testing new file carving techniques. Since current file forensics tools are efficient and capable of recovering most contiguous files, future research into file forensics will be in creating faster tools, creating new methods, adapting to new forensic issues, and recovering damaged or partial files. To assist researchers in comparing different techniques, a tool that creates test data sets is developed. This thesis describes the design decisions behind the creation of the tool, Whetstone, and the implementation of features.

Section 3.1 describes current available file forensics test data and current tools available for the generation of forensic test data. Section 3.2 describes the goals of the new test data tool and the key features required to achieve those goals. Chapter 4 discusses implementation of the tool.

3.1 Related Work

This section describes the current availability of data sets to file forensic researchers and the tools available for creating new data sets. The majority of test sets available to researchers are copies of previously used hard drives. These hard drives provide realistic data sets for a small investment and can provide very different types of data.

3.1.1 Real Used Hard Drives

The easiest way for a researcher to access a realistic hard drive for testing is to purchase a few used hard drives. In a 2007 study Garfinkel found 324 drives containing data out of 449 drives with file systems that were collected online. These drives were smaller in size (roughly 10KB to 20GB) but the type of data they contain can be very useful for testing file carvers. These 324 drives resulted in over 2 million recoverable files (about 892GB of data). Unfortunately, for researchers focusing on reassembling fragmented files only 6% of recoverable files were fragmented. While roughly half of the drives did not have any fragmentation, roughly one tenth of the drives had over ten percent fragmentation. A more detailed description of the number of fragmented drives can be found in Table 3.1. File system plays a significant role in the amount of fragmentation in a drive. The majority of disks were using the FAT file system which had

Table 3.1: Fragmentation distribution in drives [9]

Fraction of files on drive that are fragmented	Total drives	Total named files
f = 0.00%	145	17 267
$0 < f \leq 0.01$	42	459,229
$0.01 < f \leq 0.10$	107	1,115,390
$0.1 < f \leq 1.0$	30	412,297
Total	324	2,004,183

a lower fragmentation rate (about 3% of FAT drives were fragmented). NTFS and UFS file systems were more fragmented but also had a smaller sample size. They had about 12% and 16% fragmentation respectively.

The Real Data Corpus [8] is designed to give researchers access to images of real drives without having to find and purchase the right drive. It exists to aid in the creation of forensic data acquisition tools and document translation software. The collection of drives at Real Data Corpus consists of 1,289 hard drives, 643 flash drives and 98 CDs. The Corpus totals approximately 70 TB of information. It is available to 'authorized researchers' in AFF and E01 format. "In general, use of the RDC is limited to bonafide researchers operating under the oversight of an Institutional Review Board that has a DoD Assurance" [8]. While the Real Data Corpus is a useful resource, the hard drive images contain some personal data that the original owners may not want freely available. To protect those individuals' privacy the Real Data Corpus requires permission to use and is not freely available as an entry level resource for testing forensic algorithms and methods.

3.1.2 Computer Forensic Reference Data Sets

The National Institute of Standards and Technology maintains Computer Forensic Reference Data Sets (CFReDS) at www.cfreds.nist.gov. These data sets are designed for testing the effectiveness of file forensic tools. The following data sets were available in March of 2013.

- **Hacking Case** - Disk image of a laptop suspected of being used in a credit card hacking scheme
- **Russian Tea Room** - Reconstruct a menu written in Russian
- **asb image, dd, E01** - Unicode string search in Russian

- **Basic Mac Image** - Various images on a Mac File System
- **Rhino Hunt** - A small image file and network trace with hidden information about Rhinos
- **Memory Images** - Five images of live memory from different machines
- **DCFL** - Basic files for testing file recovery
- **Mobile Device Images** - A collection of images of Mobile devices. Each contains some files and information to be recovered
- **Container Files** - Container and Nested Container files in an image. String recovery tests correct manipulation of the containers
- **Deleted File Recovery** - Images to be recovered using techniques that operate on residual metadata information

The Computer Forensic Reference Data Sets also contain several disk images to test file recovery (roughly six). If a user is interested in creating additional data sets to test against, instructions for creating additional data sets are available. The instructions from CFREDs detail the process of formatting a drive and outline the steps for manipulating data files and are as follows [20]:

- Run the *not-used* program to mark each sector of a device.
- Format the device with one or more partitions of the same family.
- Synchronize the drive state by unmounting all partitions. This ensures that the current state of the drive is on the drive with no parts of the drive state only in memory.
- Image the drive to capture the base state of the formatted file system. The base image serves as a reference point to identify the initial state of file system metadata.
- Mount the file systems. The file systems are now ready to be manipulated in a controlled manner. File operations need to be grouped such that a smart operating system does not skip steps for efficient operation. For example, if we create a file and then delete the file, a smart OS may note that nothing needs to be written to secondary storage. This would undermine the effort to have something to actually recover. Operations are

grouped into sets of actions such that no action should modify the result of another action within the same set. Between each set of actions, file systems are unmounted, imaged and remounted. The actual state of the file systems can be confirmed by examining the image before continuing to the next set of actions.

- Use the *mk-file* program to create some files.
- Unmount the file systems, image and remount.
- **Do additional actions (create and append) to achieve the relationship between data blocks and metadata required for the specific test image.**
- Use the *fana* program to characterize every file to be deleted.
- Set MAC times for every file to be deleted.
- Unmount, image and remount.
- Record MAC times for every file to be deleted.
- Delete the files.
- Unmount and image the final state of the device. This final image is the test image.

The above steps provided by NIST outline the process for creating an image of a device for use with file forensic tools. The majority of steps are detailed, including setting and recording MAC times, the file systems metadata for when files are modified. The bolded step 'Do additional actions (create and append) to achieve the relationship between data blocks and metadata required for the specific test image,' is of key interest to this thesis. The instructions do not go into more detail on how to achieve the correct relationships between data blocks, which is a problem that Whetstone solves.

3.1.3 FTK Imager

FTK Imager is a forensic image creation tool from AccessData. It's primary uses are examining the contents of an image file and creating a new image file from a specified source [1]. FTK Imager (hereby referred to as Imager) does not have the same functionality for examining images as FTK the forensic toolkit, but is a free product. Imager is capable of creating new forensic images from specified sources. These sources are: physical drive, logical drive, image file, contents of a folder, and fernico device (multiple CD/DVDs). In addition to being able to create an image of disk, Imager is able to create images based off of user specified files [1]. Users can also create a set of rules for Imager to follow when adding files to an image. These rules dictate what types of files are added based on file name. For example: adding all of the .doc files in a directory and it's subdirectories. It is important to note that these rules do not apply to elements within a file, just the file name and path. Another feature of Imager is the ability to view MD5 hashes upon image creation in order to verify that the image was correctly made. These features, within a free imaging tool, serve as some base functionality for data set creation tools.

3.1.4 Fragmentation Tools

One of the aspects of Whetstone, that is unlikely to be contained in other tools, is that it should be able to reproduce fragmented files. File fragmentation is not usually encouraged, and file systems only use fragmentation when necessary. There are a few methods for purposely introducing fragmentation in a file system. A tool called MyFragmenter uses built-in windows utilities to fragment files into as many as 1000 fragments [13]. The file is split into N fragments, each the same size as the others. The fragments are placed on the disk by finding the current largest group of free clusters and placing the fragment directly in the center and repeating, until all of the fragments are allocated [13]. This method does serve to fragment the file, but it does not represent a believable scenario for a disk image. MyFragmenter was designed for use in testing disk defragmentation tools. The tool is packaged with the defragmentation tool 'MyDefrag'.

Fragger, by Passmark, has more advanced control over the type of fragmentation applied to a file. In addition to selecting the number of fragments to split a file into, Fragger allows the user to select how the fragments are spread on the disk (it contains the options: Concentrated, Scattered, Random, First Fit) [21]. Fragger also displays a basic graphical representation of how the fragments are going to be spread throughout the disk. This allows the user to quickly assess the amount of fragmentation on the drive. Designed to help test defragmentation tools, Fragger's features and user interface can be incorporated into forensic testing tools.

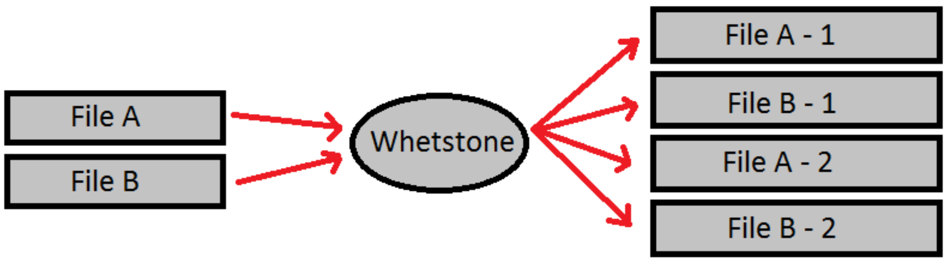


Figure 3.1: Using Whetstone to fragment files.

3.2 Goals for Whetstone

This thesis describes a new tool for creating highly specialized data sets for file forensic tool research. Since these data sets are intended to be created in order to improve file carvers, like Scalpel, and sharpen investigator skills, the tool is named Whetstone. The goals of the tool and the reasoning behind them are described in this section. Figure 3.1 shows one of the intended uses of Whetstone, creating a fragmented data set from whole files. Section 3.2.1 explains the motivation behind the tool and its primary objectives. The next section describes user functionality requirements. Lastly, Section 3.2.3 details several of the key features for Whetstone.

3.2.1 Primary Goals

Whetstone is being created so that researchers investigating file forensics problems can create data sets that are more customizable than real world data, and adaptable to problems outside of recovering different file types. In order to be more functional than currently existing tools, Whetstone needs to be able to allow for customizable file fragmentation parameters. Allowing a user to choose where and how files are fragmented will make the tool useful for research into areas regarding fragmentation. Changing the order of fragments can have an effect on the identification and classification algorithms used in file recovery. The manner in which a file is fragmented can be customized in order to emulated destroyed/corrupted fragments or mimic anti-forensic techniques. To increase its usefulness to forensic researchers, Whetstone should attempt to verify that the created image is correct. The primary goal of Whetstone is to allow file forensic researchers to conveniently create forensic data sets containing forensic files. The goal of validating the output of the tool is encompassed in this goal, in that forensic researchers need verified data sets.

Whetstone should also serve as a way for researchers to be able to explain their data sets for others to reproduce. Since forensic test cases are typically created by a researcher, the ability for another researcher to duplicate the experiment is based on the first researcher producing their data set. Whetstone should provide an alternative to hosting the original data (which can be very large). A textual, human-readable description of the data set should provide researchers the ability to communicate the details of a data set.

3.2.2 User Functionality Requirements

This section describes basic user functions that Whetstone needs. Users need to be able to select the specific files to be used by Whetstone. This allows users to control what types of files the data set will use. Since files are selected on an individual basis, preprocessing steps like moving all of the test files to a directory or special drive are unnecessary. Whetstone is required to have a graphical user interface. There are complicated methods for creating the types of data sets generated by Whetstone. Without a graphical user interface, Whetstone will not be a convenient way to create the forensic data sets and visualize what the user is creating. Visualizing the image that is going to be created by the tool should be possible in both a textual and graphical manner. A textual representation of the created image will allow a user to check certain expected results by hand (looking for patterns at a specific fragment edge). The graphical representation should make it easier for a researcher to quickly verify that the fragments are set up in the intended manner.

3.2.3 Desired Features

This section describes additional features that Whetstone is designed to contain.

- **Split a file into fragments based on fragment size, magic numbers and regular expressions from the front, end or at every occurrence.** These splitting options give users control over how the final data set will look. The methods of splitting a file are based on key elements that file carvers look for when recovering files. Splitting a file based on fragment size should help users focused on creating a specific layout and splitting based on magic numbers and regular expressions should help create data sets that focus more on the edges of fragments.

- **Group fragments into manageable sections.** This feature reduces the visual clutter that a user experiences when using Whetstone. Grouped fragments also allow the user to understand sections of fragments and categorize them.

- **Easily reorder fragments and files.** The ordering of fragments can have a significant impact on the carver so researchers are likely to create multiple data sets with the same fragments, but in a different order. Whetstone needs to make it easy for users to rearrange fragments so that these data sets can be made with the tool.

- **Remove fragments and files from the data set.** This allows users to not only remove files and fragments that are accidently added to a data set but gives users the ability to simulate fragments in slack space or fragments of a file that is not entirely in the data set.

- **Quickly reproduce and share data sets.** Whetstone is aimed at making research in file forensics more accessible. By making it easy for users to reproduce and share the data sets they create it makes it easier for potential researchers to approach test forensic software, tools and techniques.

CHAPTER 4
IMPLEMENTATION

This chapter discusses the decisions made while implementing Whetstone in Python. Whetstone was created with a focus of allowing users to easily create and manage fragments within a data set. The following sections describe how Whetstone's unique set of options and features give users control over fragmentation in the data set. Section 4.1 details the core components of Whetstone that manipulate fragments. Functions and options to manage fragments and increase ease of use for the user are described in Section 4.2.

4.1 Core Fragmentation Features

4.1.1 Fragment Size

The primary use of Whetstone is to fragment a file in order for a file carver to attempt to identify or classify fragments. File fragmentation in Whetstone is defined by the user, and fragmentation can occur for multiple reasons. Files in Whetstone can be fragmented based on the size. A user can split a file based on either the size of the new fragment or the size of the remaining fragment. This allows users to be able to emulate the methods that a file system uses when fragmenting files based on an understanding of how the file system behaves. Users can also choose to split a file into a number of fragments of a specific size, or a specific number of fragments of equal size. In situations where the split cannot occur evenly, the last fragment will be a different size from the other fragments. This option is available so that users who want to split many files into several smaller sizes can do so in an efficient manner. The final way that users can create fragments is a keyword-based approach to fragmentation. A user specifies a keyword and wherever the keyword is encountered, the current fragment is split so that everything leading to the keyword is one fragment and everything after the split is in another fragment. The idea is that users who wish to fragment files based on file format information, like header and footer information, are able to quickly do so. Fragment manipulation is based on a per fragment basis so users are able to split one file into 3 equal fragments and another file into 4 equal fragments. The multiple methods for creating new fragments in Whetstone gives a large amount of control for the user in creating the ideal data set for use.

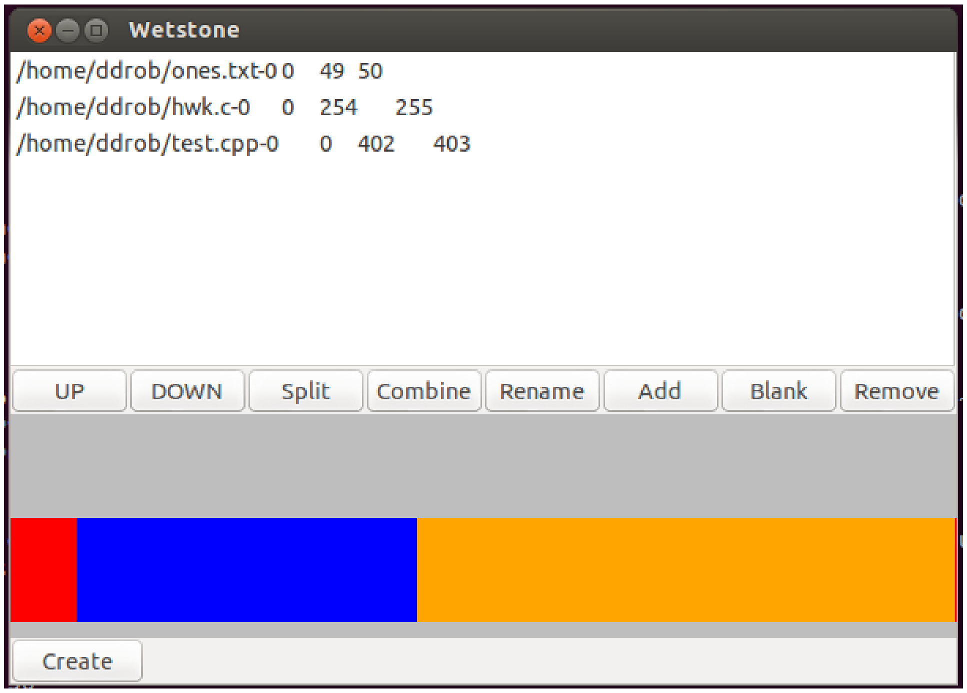
4.1.2 User Created Fragments

In addition to fragmenting existing files, Whetstone users are able to create fragments as they develop data sets. Users can add fragments consisting of purely zeros, purely ones, or random bits. These fragments are based off of ‘dev/zero’ and ‘dev/random’. The primary uses for these custom sections are for filling in fragments and creating buffers. A user emulating a specific file system environment can use created fragments to pad fragments and emulate slack space that is wiped clean or contains random bits. Fragments of ones or zeroes can be used as buffers between fragments to further distinguish between fragments. This would be useful in situations where a user is not concerned with detecting fragments, but is concerned about working with the fragments. Fragments consisting of random bits can be used as noise to increase the difficulty for tools working on the data, or to more closely emulate a realistic environment. The option of allowing a user to specify a byte sequence to use as filler was considered but ultimately this added additional complexity to a tool designed to be easy to use. An advanced user is still capable of creating a file consisting of the filler byte(s) and then places fragments of it into the data set.

4.1.3 Additional Options

Designed with researchers in mind, Whetstone contains features that are unnecessary in commercial applications. Users are given the ability to randomize the order in which fragments are written to the data set. This allows users to easily create data sets that test on the content and not on the order in which the fragments are found. Fragment classification and identification methods can be affected by fragment order and some methods use fragment adjacency as an attribute. The option of randomizing user created fragments gives users a large amount of control over the data set while still being able to quickly create slightly different data sets.

Users are also given the option of writing the data set onto a drive or into a file. Whetstone is capable writing directly to a drive (like a USB flash drive) that can be used in conjunction with tools that rely on scanning a drive for information. This option saves the user from difficult calculations that can occur when manually creating a data set with complex fragmentation. Whetstone does require that the drive is formatted with a file system that does not have optimization active. This is an unavoidable limitation because slight differences in optimization can affect the resulting data set. In instances where scanning an entire drive is not necessary,



```
/home/ddrob/ones.txt-0 0   49  50
/home/ddrob/hwk.c-0    0   254     255
/home/ddrob/test.cpp-0     0   402     403
```

Figure 4.1: The primary interface of Whetstone.

Whetstone is capable of producing an image file representation of the data set. This representation of a data set does not risk the results being altered by previously existing data on the device. While this representation does not include markers that can be found when using the full drive option it can still be used for data sets where the internals of files are the focus. It is important to note that data sets written directly to a drive will generate metadata based on the drive's file system and data sets written to a file do not contain directory structure or metadata.

4.2 Usability Features

Ease of use is important to the role that Whetstone fills for users. While all of the functions of Whetstone are possible through manual manipulation of files, Whetstone aims to make these tasks easier and more obtainable by users. The goal is that a high ease of use will encourage more new users and increase experimentation in the field of file forensics. Whetstone's user interface is designed to allow users to create a data set with simple controls or load an existing data set configuration and get an overview of the data set with a quick look. The primary screen of Whetstone's user interface can be found in Figure 4.1. The design goals are implemented using simple controls, some automation, colorful visualization and a log to track ones actions.

4.2.1 Fragment Management

Whetstone's fragment management is handled by simple controls: add/remove a fragment, split a fragment, merge fragments, move a fragment and rename fragments. Once added to the data set, entire files are treated as singular fragments. Fragments that are split into small fragments can be rejoined into manageable chunks using the merge feature. This allows a user to split a file into 100 fragments and then merge those fragments into smaller subgroups (e.g., sizes of 10,20,5,15,49,1). The smaller subgroups are easier to manage than 100 separate fragments but the user can still retain control over manipulation of the fragments. Fragment merging can also be used to indicate a completed section of the data set. When a user has finished setting up a specific layout for part of the data set, the fragments can be merged and renamed to indicate a completed section. Then the user can add and manipulate fragments while only managing one extra fragment (the section fragment) instead of the individual fragments it consists of.

Fragment sizes can be made easier to manage using Whetstone's options. Whetstone lets users add slack space to fragments based on cluster size at writing time. This lets users easily choose to emulate drive characteristics and slack space when using small fragments. The additional bytes filled by Whetstone can be filled with all zeros, all ones, or random bits, depending on what the user chooses. This option can be done manually in Whetstone by creating blank fragments and merging them with small fragments, but this option exists to give users a quick, safe way to automate that process. By default Whetstone does not enforce a cluster size, if the user enables clustering they must specify a cluster size.

Whetstone users are able to quickly view the layout of the data set with Whetstone's fragment graph. The fragment graph displays color coded blocks representing fragments to give the user an understanding of the ordering of fragments. The colored blocks are scaled to a size proportionate to the size of the fragment, which allows users to confirm that the data set is being designed how they intended. Figure 4.2 shows the results of the fragment graph after moving a large fragment.

4.2.2 Reproducing Data Sets

In a scientific environment, being able to reproduce an experiment is key. While researchers are able to directly copy images to another researcher reproducing the experiment, it can be difficult to obtain original data sets from certain researchers or after a period of time. Whetstone aims

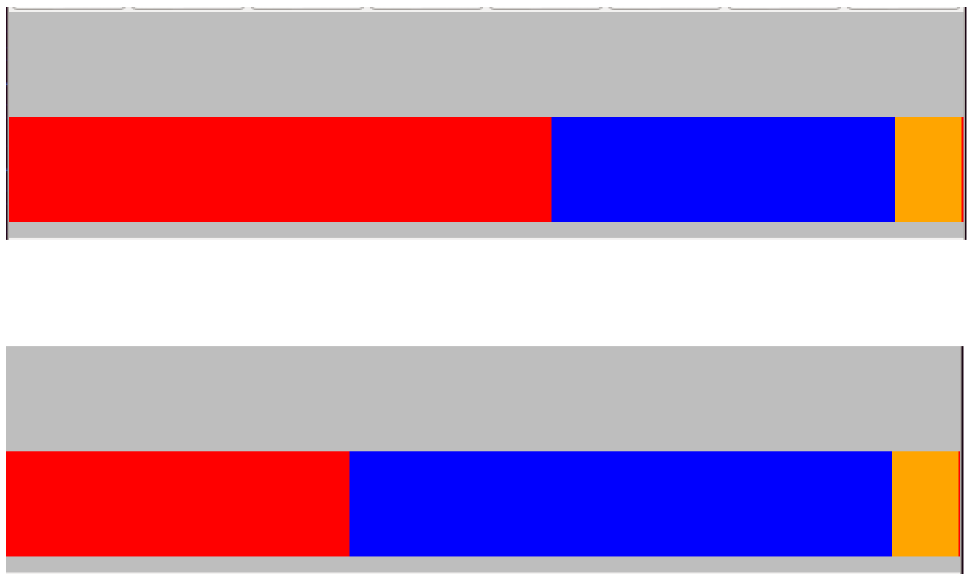


Figure 4.2: The fragment graph after moving a fragment.

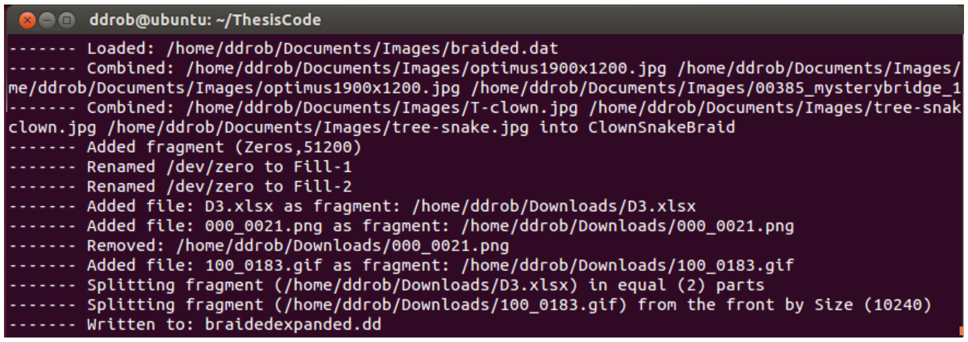
to reduce the reliance on the original experiments data set when recreating an experiment by lightweight data set layout files, data set layout tables, and by producing a reproducibility log. Users are able to seed the random number generator used by Whetstone with a specific value that can be used when recreating the data set to produce the same randomization patterns.

Like most programs Whetstone is capable of saving and loading configurations from a file. Whetstone's save and load functions do not contain the original files, which makes them smaller and easier to host and share than a file containing all of the original data. This is currently the only way to automatically recreate a data set. After loading a layout the data set can be further modified by the user. When a data set is produced using Whetstone, a table containing the data set's layout is also produced. This table contains the file name, starting and ending bytes and the size of each fragment. The table entries are arranged in sequential order. The information in the table can be used to completely reproduce a data set. A sample table produced by Whetstone can be found in Figure 4.1 (with some minor formatting).

Unlike a configuration file for the tool or a data set layout save file, the reproducibility log is designed to be read by a human and contain information about the layout creation process, not just the final layout. The reproducibility log also serves as a form of memory for the researcher when creating the data set. It is not intended to be used to automatically recreate the data set, that functionality is contained within the saving and loading of layouts. A sample reproducibility log is shown in Figure 4.3. When an interesting result from an experiment

Table 4.1: Sample data layout table

Fragment Name	Fragment Order	Start Byte	End Byte	Size
zero	0	0	51199	51200
grassbycosmic.jpg	0	0	23039	23040
connection.jpg	0	0	26076	26077
grassbycosmic.jpg	1	23040	52640	29601
zero	1	0	51199	51200
PNG-transparency-demonstration-2.png	0	0	102399	102400
honda.jpg	0	0	22312	22313
PNG-transparency-demonstration-2.png	1	102400	183959	81560



```
------- Loaded: /home/ddrob/Documents/Images/braided.dat
------- Combined: /home/ddrob/Documents/Images/optimus1900x1200.jpg /home/ddrob/Documents/Images/
me/ddrob/Documents/Images/optimus1900x1200.jpg /home/ddrob/Documents/Images/00385_mysterybridge_1
------- Combined: /home/ddrob/Documents/Images/T-clown.jpg /home/ddrob/Documents/Images/tree-snak
clown.jpg /home/ddrob/Documents/Images/tree-snake.jpg into ClownSnakeBraid
------- Added fragment (Zeros,51200)
------- Renamed /dev/zero to Fill-1
------- Renamed /dev/zero to Fill-2
------- Added file: D3.xlsx as fragment: /home/ddrob/Downloads/D3.xlsx
------- Added file: 000_0021.png as fragment: /home/ddrob/Downloads/000_0021.png
------- Removed: /home/ddrob/Downloads/000_0021.png
------- Added file: 100_0183.gif as fragment: /home/ddrob/Downloads/100_0183.gif
------- Splitting fragment (/home/ddrob/Downloads/D3.xlsx) in equal (2) parts
------- Splitting fragment (/home/ddrob/Downloads/100_0183.gif) from the front by Size (10240)
------- Written to: braidedexpanded.dd
```

Figure 4.3: A sample reproducibility log.

occurs, a researcher can look at the reproducibility log for an easy to read understanding of the design of the data set. The reproducibility log lists the actions taken by the user in chronological order. It lists the actions and the fragments they are performed on. While Whetstone does not currently contain any method for adding notes to the log during set creation, this may be added in the future. The format of the reproducibility log can be found in Appendix B which allows researchers to easily parse through the log.

4.2.3 User Guide

The use of Whetstone's features is demonstrated in Appendix A through step-by-step instructions. Additional images breakdown the aspects of the user interface and how they relate to the features described in this chapter. The user guide includes the properties and settings that can be changed in Whetstone.

CHAPTER 5
RESULTS TESTING

This section describes the tests performed on Whetstone and their results. The main goal here is to show that Whetstone is creating data sets correctly and at the same level of current available data sets. Whetstone's testing should also show the need for specialized fragmented data sets. To demonstrate this need, data sets created by Whetstone are tested with current forensic tools to demonstrate the effects of fragmentation.

5.1 Correctness Testing

To show that Whetstone is producing the expected data set, a series of tests comparing Whetstone data sets to known fragmented data sets were conducted. Whetstone was used to produce 30 data sets based on forensic images from the Computer Forensic Reference Data Sets File Carving section [20]. These images include non-fragmented files, sequential fragmentation, non-sequential fragmentation, missing files, nested files, and braided files. Non-fragmented files are whole files that have never been split. Sequentially fragmented files are files that have been split into two or more fragments and then reassembled in the original order. Non-sequential fragmentation are fragmented files where the fragments are reassembled in an incorrect order. Missing files are fragmented files reassembled in the correct order but with one or more fragments missing. Nested files are non-fragmented files inbetween a sequentially fragmented file. Braided files are two sequentially fragmented files combined so that every other fragment is the next sequential fragment. The following file types are used in the images: avi, docx, gif, jpg, mov, pdf, png, pptx, rar, txt, wav, wmv, xlsx, and zip. The Whetstone data sets were created from files downloaded from CFREDs and using the image layouts provided.

To compare the Whetstone data set to the CFREDs data set, the sets were carved with Foremost and the results of each carving were compared against each other. Since CFREDs uses xB9 as filler and Whetstone uses x00 direct comparison of the data sets was not applicable. While it is possible to create filler fragments using xB9 as the filler byte, the tests were conducted with the mindset that the filler bytes were randomized (as a pseudo-worst case scenario). The comparison tests were done between three different sources: the CFREDs data sets, Whetstone without a specified cluster size (Whetstone's close fit option), and Whetstone with Clusters of size 512. The first comparison test compares the number of files recovered from each data set.

Table 5.1: Number of files recovered by extension

Ext	CFREDs	Close Fit	Cluster Fit
avi	5	5	5
docx	8	7	8
gif	5	5	5
jpg	23	23	23
mov	12	12	12
pdf	5	5	5
png	9	9	9
pptx	5	5	5
rar	6	6	6
txt	31	31	31
wav	6	6	6
wmv	9	9	9
xlsx	8	7	8
zip	14	14	14

The number of files recovered by each extension are displayed in Table 5.1. The entries for docx and xlsx are the interesting entries with the close fit produced by Whetstone recovering one less docx and xlsx file. This is caused by the carver using the filler space to terminate a docx and xlsx. Without the filler space between a fragment and the end of the cluster/sector the file carver combines two files.

The second comparison test evaluates the contents of the recovered files. The comparison was done by calculating md5 hashes for each file and then comparing the md5 hashes. If the files differ in any way they will produce different md5 hashes. While the different filler bits (xb9 vs x00) will cause differences between files, the majority of recovered files should not include these filler sections. The data set produced by Whetstone using the close fit setting matched 70.4% of the files recovered from the CFREDs set. By choosing the cluster option the number of matches increased to 80.87%. These results are shown in Table 5.2. Ideally the Whetstone data sets should produce closer to 100% (although in the case of the close fit option it should not reach 100%). To test the files that were not identically matched, the files were individually inspected to see why they did not match. The difference were caused by the recovered files including some filler bits. Since the bits were originally different but both files had filler bits in the same location, it can be said that the files matched. It is important to note that these tests compared all both fully and partially recovered files. Since the goal of this test was to determine if Whetstone was producing the same data set as expected, the effectiveness of the

Table 5.2: Percentage of files correctly matched to CFREDs data set (including differences in filler bits)

Type	% Matched
Close Fit	70.4%
Cluster Fit	80.87%

file carving are irrelevant; it only matters that the files carved from each data set match.

5.2 Testing Effects of Fragmentation

The previous tests show that Whetstone is capable of creating highly specialized data sets for use in improving forensics tools. The tests found in this section show that these highly specialized data sets are useful and necessary for improving file carving when handling fragmented data. To show this a series of tests are conducted with the purpose to outlining the effects of a fragmented data set on the file carving process. The tests are expected to show that fragmented files are recovered at a lower success rate than non-fragmented files and that increased complexity of fragmentation decreases the success rate. Figure 5.1 depicts the process used for testing the effects of fragmentation. The figure shows data sets being created through Whetstone and then carved with two different file carvers, Foremost and PhotoRec, then the results of the carving are compared to the original files. This section details this process and the various fragmentation types that are tested.

5.2.1 Test Methodology

The results of carving files out of five different types of fragmentation are compared to the results of carving a non-fragmented file in these tests. The first type of fragmentation, non-sequential fragmentation, contains data sets where a file's fragments are out of order. The data set is constructed of images split into two, three or four fragments. The fragments are placed in forward, reverse and a scrambled order. Each of these tests is run with a cluster size of 512. Table 5.3 shows a sample of the final layout of a non-sequential data set. The fragment order column contains the number that indicates where the fragment is in the files original order. For example the original T-clown.jpg can be formed by assembling the fragments in this order: T-clown.jpg (0), T-clown.jpg (1), T-clown.jpg (2).

The second and third type of fragmentation are based on a file placed between fragments

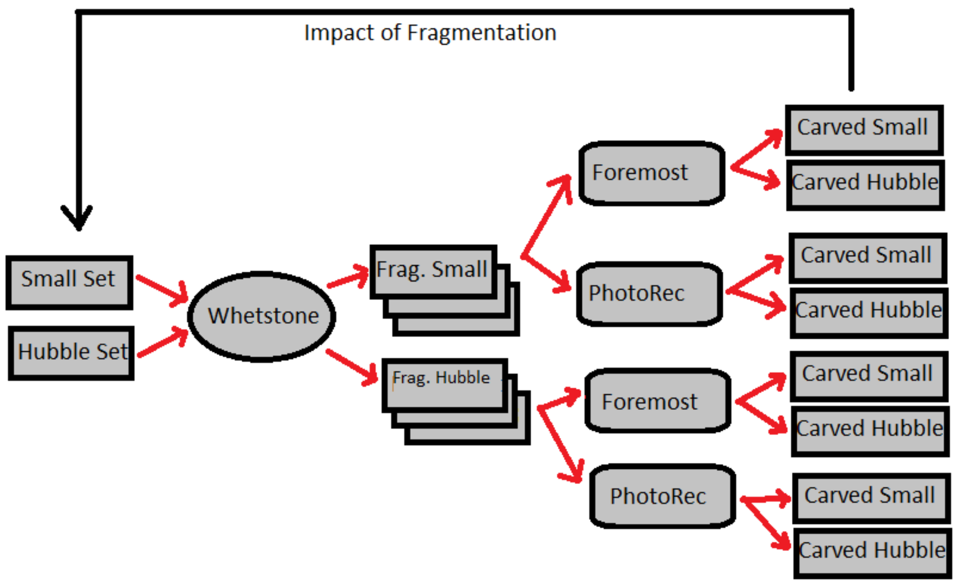


Figure 5.1: Procedure for testing the effects of fragmentation.

of another file. Data sets where the fragments surrounding a file are sequential, referred to as Nested Files, are demonstrated in Table 5.4. The key feature of this data set is demonstrated between connection.jpg and grassbycosmic.jpg. A complete file, connection.jpg, is surrounded by two fragments belonging to grassbycosmic.jpg. It is important to note that the fragments of grassbycosmic.jpg are arranged so that the fragments are in ascending order. This is different from the third type of fragmentation, Nested non-sequential fragmentation. Shown in Table 5.5, nested non-sequential fragmentation still has a complete file surrounded by fragments. Unlike nested fragmentation these surrounding fragments are not in ascending order. We can see that connection.jpg surrounds reaching.jpg and starts with fragment 1 and ends with fragment 0. Nested non-sequential data sets also have multiple fragments surrounding a file (at most two fragments per side). This is demonstrated between overseer.jpg and Apple-safari.png. The additional fragmentation greater tests the file carvers ability to handle non-sequential fragments. Data sets containing nested files and nested non-sequential files are both used in testing.

Data sets were also constructed that consisted of fragments nesting around another file fragment. Unlike nested and nested non-sequential fragmentation based data sets, the data set for nested fragments does not use a complete file as the center for a nest. Table 5.6 has a sample of the nested fragment data sets used. The data set consists of sequential fragments surrounding a central fragment. The data sets used contain central fragments that are either

Table 5.3: Non-sequential data set layout

Name	Fragment Order	Start Byte	End Byte	Size
zero	0	0	51199	51200
tree-snake.jpg	1	25600	54083	28484
tree-snake.jpg	0	0	25599	25600
T-clown.jpg	0	0	15359	15360
T-clown.jpg	2	30720	52120	21401
T-clown.jpg	1	15360	30719	15360
Apple-Safari.png	0	0	15359	15360
Apple-Safari.png	2	30720	46079	15360
Apple-Safari.png	1	15360	30719	15360
Apple-Safari.png	3	46080	64126	18047
V-for-Vandetta-1920-4.jpg	0	0	20479	20480
V-for-Vandetta-1920-4.jpg	1	20480	42782	22303

the starting fragment (connection.jpg) or a fragment from the middle of a file (honda.jpg). The surrounding fragments in the test data sets are all in ascending order.

The final data set used is based around braiding file fragments together. In a braided data set, two files are sequentially fragmented and are placed in alternating order. Table 5.7 shows a sample of the braided data set. The sample shows braiding between the files optimus.jpg and mysterybridge.jpg and braiding between T-clown.jpg and tree-snake.jpg. The data sets constructed for testing have each file's fragments in sequential order. This is because the braided test is designed to test the ability of a file carver to correctly determine the edges of a fragment and is not designed to test reordering fragments. The constructed data sets heavily feature jpg and png files and were created using Whetstone. When conducting the tests, the criteria for successfully carving a file is based on the carved files ability to be validated by a human. Essentially, each file can be opened and looks identical to the original.

To highlight the effects of fragmentation on the data sets, two carvers are used: Foremost and PhotoRec (discussed in Section 2.4.1). Foremost is a file carver that reassembles files from header and footer information as it comes across the files. Foremost does not use specific algorithms to handle fragmentation. Tests using Foremost are designed to show the difference between a generic file carver and a specialized file carver. For these tests the Foremost configuration file has all file types on and is being run on Linux [3]. The second carver, PhotoRec, is a file carver that was originally designed to recover images from digital camera disks [6]. PhotoRec uses header and footer information to determine cluster size and determine fragments.

Table 5.4: Sample of a nested data set

Name	Fragment Order	Start Byte	End Byte	Size
zero	0	0	51199	51200
grassbycosmic.jpg	0	0	23039	23040
connection.jpg	0	0	26076	26077
grassbycosmic.jpg	1	23040	52640	29601
zero	1	0	51199	51200
PNG-transparency-demonstration-2.png	0	0	102399	102400
honda.jpg	0	0	22312	22313
PNG-transparency-demonstration-2.png	1	102400	183959	81560
zero	2	0	51199	51200
spm-1920.jpg	0	0	43519	43520
Apple-Safari.png	0	0	64126	64127
spm-1920.jpg	1	43520	78513	34994

It uses validation techniques to determine if a fragment belongs to a file. These tests use PhotoRec's unknown file system setting and PhotoRec is set to keep corrupted files (which is used to determine incomplete files).

5.2.2 Results

The tests in this section are performed with the two file carvers, Foremost and PhotoRec, over the six types of data sets (sequential, non-sequential fragments, nested, nested non-sequential, nested fragments and braided). For each data set type there are two tests run. One test is performed over a small data set that matches the sample layouts discussed in Section 5.2.1 and a larger data set constructed from 30 images taken by the Hubble Telescope [2]. The smaller data set (sample data set) uses both jpg and png files from various internet sources and the Hubble data set uses only the jpg file format. A summary of the results on the Hubble data set can be found in Table 5.9.

Sequential Fragments: When dealing with sequential fragments both file carvers were able to correctly recover all of the files in the small data set. Since this is the simplest case these results are expected. On the Hubble data set Foremost recovered all 30 image files and PhotoRec fully recovered 28 images and 2 corrupted jpg files. This is slightly unusual since PhotoRec is designed to recover images and Foremost is more suited to multiple file types. The two corrupt jpg files are the result of PhotoRec recovering files cluster by cluster. This means that unlike files recovered by Foremost, PhotoRec includes slack space in recovered files in circumstances

Table 5.5: Nested non-sequential fragmentation

Name	Fragment Order	Start Byte	End Byte	Size
zero	0	0	51199	51200
gwp-smgalaxy.jpg	1	51200	91744	40545
barcelona.jpg	0	0	79444	79445
gwp-smgalaxy.jpg	0	0	51199	51200
zero	1	0	51199	51200
connection.jpg	1	10240	26076	15837
reaching.jpg	0	0	46042	46043
connection.jpg	0	0	10239	10240
zero	2	0	51199	51200
overseer.jpg	0	0	9215	9216
overseer.jpg	2	18432	27647	9216
Apple-Safari.png	0	0	64126	64127
overseer.jpg	1	9216	18431	9216
overseer.jpg	3	27648	36975	9328

where the slack space does not greatly increase the size of the file. In the Hubble data set, the excess bytes in the final cluster cause PhotoRec to recover a corrupt JPG file. These tests serve as a baseline of expectations for the files.

Non-Sequential Fragments: The tests with non-sequential fragmentation are more interesting. In the small data set, whose layout is shown in Table 5.3, Foremost was able to partially recover tree-snake.jpg. PhotoRec was able to partially recover two fragmented jpgs (tree-snake.jpg and T-clown.jpg), although T-clown.jpg was being reassembled in an incorrect order and the image wrapped around itself. In both cases the partially recovered files are recovered from the start of the images and are missing fragments that don't fall between a jpg header and footer. This shows that the ordering of fragments will play a heavy role in the effectiveness of a file carver.

The difference between Foremost and PhotoRec is exemplified in the Hubble test set with non-sequential fragmentation. Foremost recovered 22 partial images where PhotoRec recovered 28 partial images (and 2 corrupt images). Here we can see that a PhotoRec's fragmentation reassembly algorithms are an improvement over Foremost.

Nested Files and Fragments: The possible effects of a single fragment are demonstrated by the tests on data sets including nested files and fragments nested within a file. The small data sets test data are shown in Table 5.4 and Table 5.6. The difference between these two data sets is that the inner files are complete in the nested set and are fragments in the nested fragment

Table 5.6: Nested fragment fragmentation data set

Name	Fragment Order	Start Byte	End Byte	Size
zero	0	0	51199	51200
grassbycosmic.jpg	0	0	23039	23040
connection.jpg	0	0	10239	10240
grassbycosmic.jpg	1	23040	52640	29601
zero	1	0	51199	51200
PNG-transparency-demonstration-2.png	0	0	102399	102400
honda.jpg	1	10240	22312	12073
PNG-transparency-demonstration-2.png	1	102400	183959	81560
zero	2	0	51199	51200
spm-1920.jpg	0	0	43519	43520
Apple-Safari.png	1	15360	30719	15360
spm-1920.jpg	1	43520	78513	34994

Table 5.7: Braided fragmentation data set

Name	Fragment Order	Start Byte	End Byte	Size
zero	0	0	51199	51200
optimus.jpg	0	0	20479	20480
mysterybridge.jpg	0	0	40959	40960
optimus.jpg	1	20480	56179	35700
mysterybridge.jpg	1	40960	80432	39473
T-clown.jpg	0	0	20479	20480
tree-snake.jpg	0	0	20479	20480
T-clown.jpg	1	20480	52120	31641
tree-snake.jpg	1	20480	54083	33604

set. On the small nested data set Foremost was able to completely recover two of the three inner files and recovered the start of two outer files. Figure 5.2 shows a partially recovered outer file. PhotoRec's fragment reassembly techniques work on the nested data set and PhotoRec was able to correctly recover all inner and outer files. Fragmenting the inner image had a significant impact in the results. The best case scenario for this test is three complete files (outer) and three fragments (inner). Foremost was able to partially recover the same outer files as in the nested set but did not recover any of the inner files. The results of fragmenting the nested file also decreased PhotoRec's recovery, with each outer file only being partially recovered and a single inner fragment being recovered.

The difference in the effectiveness of the file carvers is apparent in the results of the nested file and nested fragmentation tests on the Hubble data set. In the Hubble nested file data set

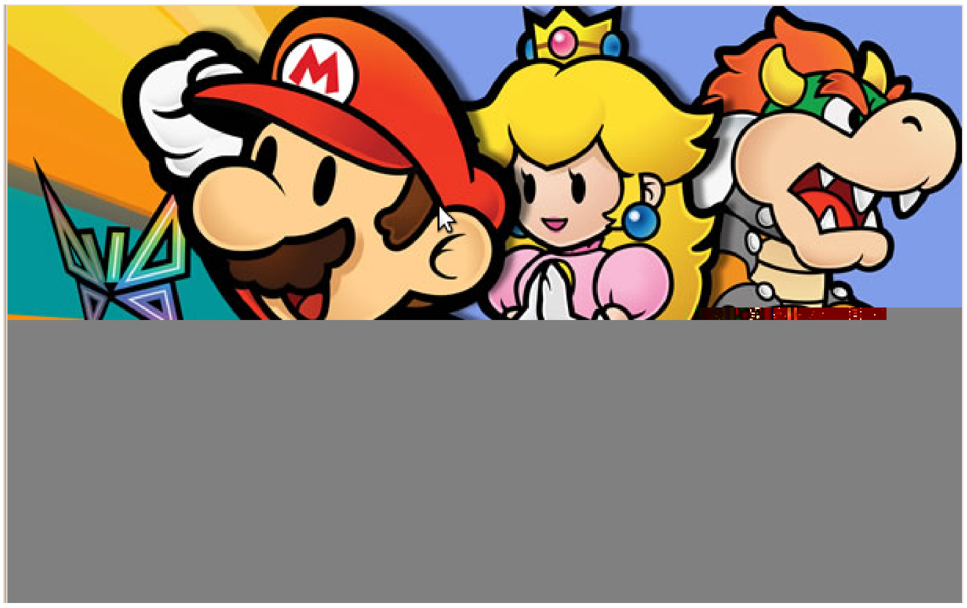


Figure 5.2: A partially recovered file.

Foremost recovered 15 partial images and PhotoRec recovered 16 full images, 12 partial images and 2 corrupted jpg files. In addition to the 12 partial images recovered, 8 of them were visually correct except the bottom half of the files were tinted (one could argue that these files were correctly recovered). After fragmenting the nested files, Foremost partially recovered 15 files and PhotoRec recovered 28 partial images. This pairing of tests is significant for two reasons. First these tests show the difference even simple fragmentation algorithms can play in the recovery of files. Foremost's algorithms did not attempt to recover the inner files/fragments. Second these tests show that a rogue file fragment can cause a significant decrease in the effectiveness of a file carver.

Nested Non-Sequential Fragments: The layout for the small nested non-sequential data set is shown in Table 5.5. On this data set Foremost behaved as expected with each inner file being correctly recovered and the outer files being only partially recovered. Foremost did not partially recover all of the outer fragments. Interestingly it did not recover overseer.jpg which started with the file header (Fragment order: 0,2,1,3). This was unexpected since Foremost has attempted to recover any files with a proper header in the other tests. PhotoRec's performance on the small nested non-sequential data was unexpected. Like Foremost, PhotoRec correctly recovered inner files and partially recovered fragments of the outer files. What is interesting about PhotoRec's recovery is that the non-sequential file starting with the header was recovered

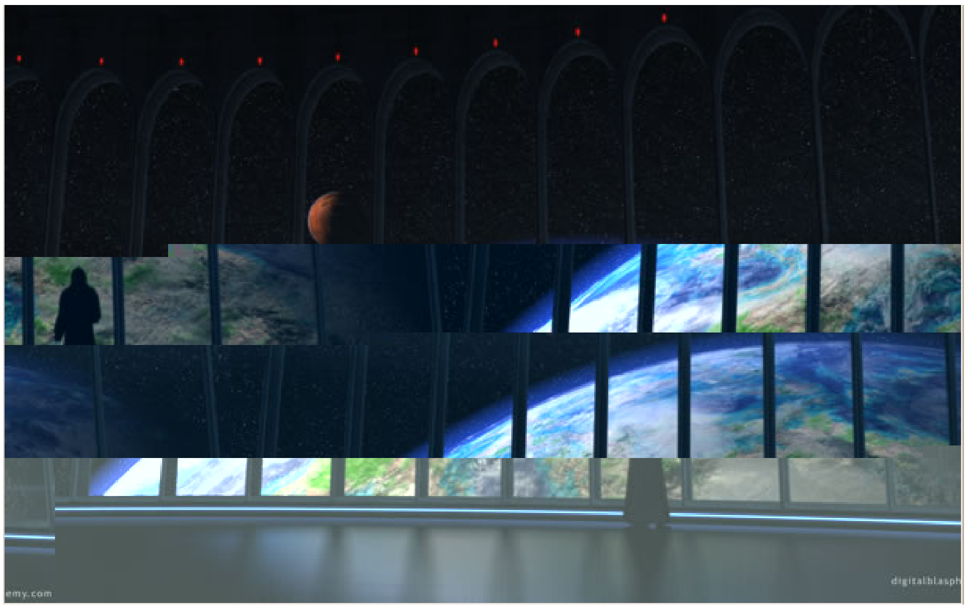


Figure 5.3: A misordered image

although the picture is out of order (but the entire image is there). This recovered image is shown as Figure 5.3. This demonstrates that the ordering of fragments is unimportant to PhotoRec but any fragments encountered before an incomplete header fragment are ignored and that jpg image files are structured in such a way that rearranged data blocks can still produce 'valid' incorrect files. How would this data set be recovered by a file carver that worked from back to front?

The Hubble data set helps demonstrate the problems that can arise when headers and footers are encountered out of order. The non-sequential nested data set includes an even number of file headers and footers but it is still possible for two headers to be encountered before a footer. Situations like this decrease the effectiveness of Foremost which recovered 7 full images and 11 partial images. PhotoRec was able to reassemble 14 full images and 14 partial images. Fragmentation algorithms like the one used by PhotoRec can decrease the amount of data thrown out by the carver and can provide significant improvement in results.

Braided Fragments: The last and most complex data set is also the most troubling. The braided data set takes two bi-fragmented files and alternates between them (Sample layout shown in Table 5.7). On the small braided data set Foremost's performance was poor. Half of the files were able to be partially recovered. The recovered images show that the recovered fragments were the first fragments with a header that appear after a fragment with footer

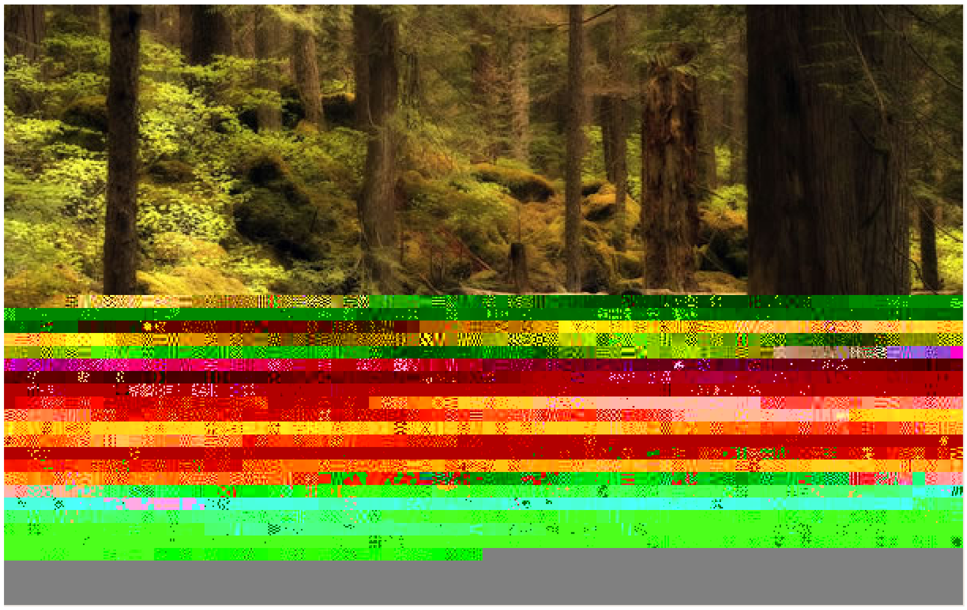


Figure 5.4: A braided image recovered by PhotoRec.

information (or the very first header fragment). Recovering part of every other file does not seem to be a very good result. PhotoRec's results are better with a partial recovery for each file. The images recovered by PhotoRec match each header fragment with the first footer fragment encountered. Initially this seems like a fairly good result, each fragment was recovered they just got a little jumbled. Figure 5.4 is one of these jumbled images, it contains the header fragment for a picture of a forest scene and the footer fragment for a picture of Optimus Prime. The problem is that determining which file each fragment belongs to is a difficult task and a braided file system would not be impossible. The results of all of the small data set tests are shown in Table 5.8

The findings from the tests on the small braided data sets are confirmed in the tests on the braided Hubble data set. Foremost recovered 15 partial images and PhotoRec recovered 28 partial images and 2 corrupted jpg files. Considering the increase in carving difficulty from braided files, a file system based around braiding files would pose a significant threat to current file carving techniques. The results of all of the Hubble data set tests are shown in Table 5.9.

Table 5.8: Files recovered from the small data sets

-		Foremost		PhotoRec	
Type	Possible Files	Whole File	Partial File	Whole File	Partial File
Sequential	4	4	0	4	0
Non-Sequential	4	1	2	1	3
Nested	6	2	2	6	0
Nested Fragment	6	0	2	0	4
Non-Sequential Nested	6	3	2	3	3
Braided	4	0	2	0	4

Table 5.9: Files recovered from the Hubble data set of 30 images

-	Foremost		PhotoRec		
Type	Whole File	Partial File	Whole File	Partial File	Corrupt File
Sequential	30	0	28	0	2
Non-Sequential	0	22	0	28	2
Nested	0	15	16	12*	2
Nested Fragment	0	15	0	28	2
Non-Sequential Nested	7	11	14	14	2
Braided	0	15	0	28	2

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Conclusion

Computer forensics tools are used to analyze cyberattacks, recover deleted files, track data, detect malware and gather evidence in criminal investigations. File recovery tools are used to extract data from disk drives that are intact, corrupted or damaged. File recovery tools called file carvers are able to recover files without using file system metadata. File carvers are most commonly used to recover files from unknown file systems and drives with damaged sectors. One of the obstacles that file carvers face is file fragmentation. A fragmented file is split into multiple pieces throughout the file system.

There are many different forms of file carvers, some handle fragmentation better than others. The goal of this thesis is to design a tool that creates fragmented data sets that file carvers can test against. By creating fragmented data sets, the effects of fragmentation and how file carvers handle different fragmentation scenarios can be studied. This tool, Whetstone, is designed to give users more control over the creation of data sets than current data set creation tools.

Whetstone's features focus on user control on where to fragment a file, manipulating the order of fragments, visualizing the data set and reproducibility. Whetstone gives users the ability to split a file based on multiple criteria and recombine fragments for easier management. Users are also able to freely rearrange fragments and view the layout at all times. To make sharing data sets easier Whetstone uses both a save/load function and a reproducibility log that let other users expand and revist previous data sets.

In this thesis, the effects of fragmentation are demonstrated on data sets created by Whetstone. They show that rogue fragments can heavily disrupt a file carver and that simple fragmentation patterns (like alternating fragments) can prevent proper recovery of files. Whetstone's correctness is also tested and compared to data sets for forensic testing provided by NIST (National Institute of Standards and Technology).

In summary, the work presented in this thesis designs and implements a tool for creating user specified data sets for forensic tool testing.

6.2 Future Work

The previous section has summarized the work of this thesis. However, as in every research effort, there still are a number of areas where further work could lead to other important conclusions. Some of these areas include:

1. Increase ability to mass produce data sets

Currently Whetstone is designed around creating a very specific data set in order to test a specific aspect of a tool or recreate a specific situation. Whetstone is not designed to make the creation of generic data sets easier. Functionality could be added to facilitate in the creation of general data sets so that users do not need to use two methods of data set creation.

2. Usability improvements

As with any software product there is room for improvement based on user feedback. While Whetstone was created with ease of use in mind there are improvements that can be made to make the user experience even easier.

3. Make Whetstone usable with Windows

There are several file carving tools designed for use with the Windows operating system. By making Whetstone available on more than just the Linux operating system the tool is more likely to be used.

4. Tool Security Adjustments

Whetstone has several ways for a user to execute shell commands. There are possible configurations of user accounts that could give Whetstone users the ability to execute shell commands with unintend permissions. This limits Whetstone's usefulness in multi-user environments.

5. Performance Increase

The rate at which Whetstone copies files can be very slow depending on the fragmentation setup. This is typically seen when handling a large file split into uneven fragments. It is possible that adjustments could be made to the block sizes chosen by Whetstone in order to speed up interactions with files. This would make Whetstone more attractive to users who need to perform tests with large scale data sets.

6. Error Proof Data Sets

One of the common elements in current file carvers is that the user can verify that the data being worked on is the correct data from the original device. It would make sense for Whetstone to contain a similar feature that helps assure users that the data set created was created properly. Currently this is done by using an approved method of copying but that does not provide the user with proof.

In summary there are many opportunities to make Whetstone more attractive to users working with file carvers and file recovery tools.

BIBLIOGRAPHY

[1] Accessdata product downloads. http://www.accessdata.com/support/product-downloads, 2013. “Accessed: 2012-12-10”.

[2] E. S. Agency. Images — esa/hubble. www.spacetelescope.org/images.

[3] Foremost. foremost.sourceforge.net, 2001-2013.

[4] M. Amirani, M. Toorani, and A. Beheshti. A new approach to content-based file type detection. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, pages 1103–1108, July 2008.

[5] L. Aronson and J. van den Bose. Towards an engineering approach to file carver construction. *Computer Software and Applications Conference Workshops (COMPSACW)*, pages 368–373, July 2011.

[6] CGSecurity. Photorec. www.cgsecurity.org/wiki/PhotoRec.

[7] M. Chen, N. Zheng, M. Xu, Y. Lou, and X. Wang. Validation algorithms based on content characters and internal structure: The pdf file carving method. In *Information Science and Engineering, 2008. ISISE '08. International Symposium on*, volume 1, pages 168–172, Dec 2008.

[8] Digital corpora: Producing the digital body. http://digitalcorpora.org/corpora/disk-images/real-data-corpus, 2009-2013. “Accessed: 2013-2-16”.

[9] S. Garfinkel. Carving contiguous and fragmented files with fast object validation. In *2007 Digital Forensics Research Workshop (DFRWS)*, pages 4S:2–12, 2007.

[10] T. Holleboom and J. Garcia. Fragment retention characteristics in slack space analysis and measurements. *Security and Communication Networks (IWSCN)*, pages 1–6, 26–28, May 2010.

[11] M. Karresand and N. Shahmehri. File type identification of data fragments by their binary structure. In *Information Assurance Workshop, 2006 IEEE*, pages 140–147, June 2006.

[12] M. Karresand and N. Shahmehri. Oscar - file type identification of binary data in disk clusters and ram pages. In *Security and Privacy in Dynamic Environments, 2006. Proceedings of IFIP International Information Security Conference*, 2006.

[13] J. Kessels. Myfragmenter. http://www.mydefrag.com/SeeAlso-MyFragmenter.html, 2009. “Accessed: 2013-2-16”.

[14] M. McDaniel and M. Heydari. Content based file type detection algorithms. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 10 pp., Jan 2003.

[15] R. McKemmish. What is forensic computing. *Trends and Issues in Crime and Criminal Justice*, 118.

[16] N. Memon and A. Pal. Automated reassembly of file fragmented images using greedy algorithms. *Image Processing, IEEE Transactions on*, 15(2):385–393, 2006.

[17] N. Memon and A. Pal. The evolution of file carving. *Signal Processing Magazine*, 26(2):59–71, March 2009.

[18] MistWiz. Disk-structure.svg. http://upload.wikimedia.org/wikipedia/commons/d/d7/Disk-structure.svg.

[19] S. Moody and R. Erbacher. Sadi - statistical analysis for data type identification. In *Proceedings of the 3rd IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 41–54, May 2008.

[20] The cfreds project. www.cfreds.nist.gov, March 2013.

[21] Fragger: File fragmentation tool. http://www.passmark.com/products/fragger.htm, 2013. “Accessed: 2013-2-16”.

[22] M. Pontello. Trid-file identifier. http://mark0.net/soft-trid-e.html.

[23] G. G. Richard III and V. Roussev. Scalpel: A frugal, high performance file carver. In *Digital Forensics Research Workshop (DFRWS)*, 2005.

[24] V. Roussev and S. Garfinkel. File fragment classification-the case for specialized approaches. *Systematic Approaches to Digital Forensic Engineering*, pages 3–14, 21–21, March 2009.

[25] K. Shanmugasundaram and N. Memon. Automatic reassembly of document fragments via context based statistical models. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 152–159, Dec 2003.

[26] J. V. Sid Leach and A. Bishop. What every lawyer needs to know about computer forensic evidence. *Snell & Wimer LLP*, 2010.

[27] L. Sportiello and S. Zanero. File block classification by support vector machine. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 307–312, Aug 2011.

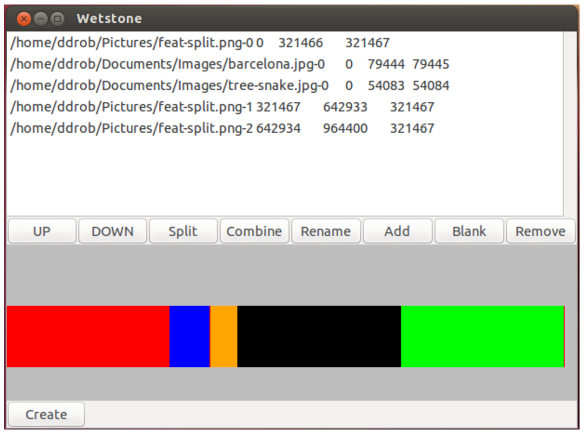
[28] V. Thing, T.-W. Chua, and M.-L. Cheong. Design of a digital forensics evidence reconstruction system for complex and obscure fragmented file carving. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 793–797, Dec 2011.

[29] C. Veenman. Statistical disk cluster classification for file carving. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 393–398, Aug 2007.

[30] J. Weston. Support vector machine (and statistical learning theory). http://www.cs.columbia.edu/~kathy/cs4701/documents/jason_svm_tutorial.pdf.

APPENDIX A

Whetstone User Guide



```
/home/ddrob/Pictures/feat-split.png-0 0    321466     321467
/home/ddrob/Documents/Images/barcelona.jpg-0     0    79444  79445
/home/ddrob/Documents/Images/tree-snake.jpg-0    0    54083  54084
/home/ddrob/Pictures/feat-split.png-1 321467     642933     321467
/home/ddrob/Pictures/feat-split.png-2 642934     964400     321467
```

Figure 1: Whetstone's main screen.

This section outlines the steps required to perform Whetstone's basic functions. Figure 1 shows the default screen for Whetstone. Figure 2 shows the preferences menu.

Adding a file: This adds a file to the list of fragments. The new file is added to the end of the current list of fragments.

- Press 'Add'.
- Browse to the file you wish to add.
- Select the file.
- Press 'Open'.

Removing a fragment: This removes a fragment from the list of fragment. This action cannot be undone.

- Click the name of the fragment you want to remove (Use CTRL or SHIFT to select multiple fragments).
- Press 'Remove'.
- Press 'Yes'.

Moving a fragment: Reorder the fragment list.

- Select the fragment you wish to move.
- Press 'Up' or 'Down' to move the fragment forward or backward in the fragment order.

Splitting a fragment: This splits a fragment into at least two smaller fragments. The newly formed fragments are placed at the end of the fragment list. The original fragment (the front of the fragment) remains in the same order in the fragment list.

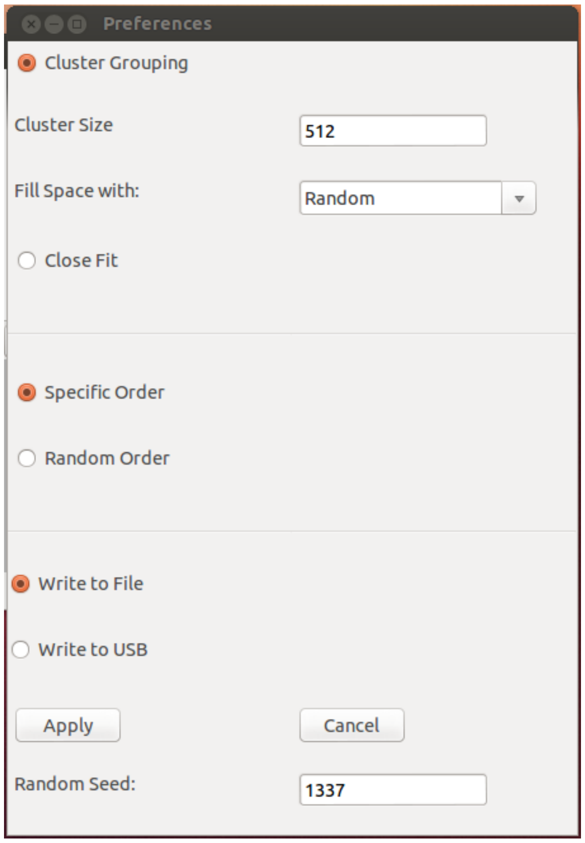
- Select the fragment to split.
- Press 'Split'
- Press the bubble corresponding to the type of split you wish to perform (size, regular expression, equal parts).
- Enter the size, regular expression or number of equal parts in the text box corresponding to the selected bubble.
- For size and regular expression choose to split based on the first occurrence, last occurrence or every occurrence. (e.g. Create a fragment from the first 512 bytes, the last 512 bytes or every 512 bytes is a different fragment.)
- Press 'Apply'.

Combine fragments: Combines multiple fragments together under one manageable name. The new fragment consists of the original fragments assembled in the fragment list order. The new fragment is placed at the end of the fragment order.

- Select the fragments to combine (use CTRL and SHIFT to select multiple fragments).
- Press 'Combine'.
- Input a label for the new fragment.
- Press 'Ok'.

Creating a fragment: Created fragments are fragments consisting of either all zeros, all ones, or a random mixture of both. Users can specify the size of the new fragment in bytes and the fragment is added to the end of the fragment order.

- Press 'Blank'.
- Select zeros, ones or random to determine the type of bits the fragment consists of.
- Press 'Ok'.
- Enter the desired size, in bytes, of the fragment.
- Press 'Ok'.



[✓ Cluster Grouping]

Cluster Size: 512

Fill Space with: Random

Close Fit

[✓ Specific Order]

Random Order

[✓ Write to File]

Write to USB

Random Seed: 1337

Figure 2: Preference menu

Saving a data set layout: This will save the current fragment order as a data set layout for future use.

- Select ‘File’.
- Select ‘Save Layout’. (Alternatively use CTRL-S)
- Choose a directory to save the layout to.
- Name the layout and hit ‘Save’.

Loading an existing data set layout: This will load a previously saved data set layout.

- Select ‘File’.
- Select ‘Open Layout’. (Alternatively use CTRL-O)
- Select the layout you wish to load.
- Press ‘Open’.

Enable Clustering: By default Whetstone uses 'close fit' which means that the fragments do not have slack space after them to fill out a cluster. Enabling cluster grouping will add zeros, ones or random bytes to the end of a fragment until it is divisible by the cluster size.

- Select 'Settings'.
- Select 'Properties'.
- Press the bubble next to Cluster Grouping.
- Input the cluster size (in bytes) you wish to use.
- Select the dropdown menu and chose what type of bit you want to fill in the slack space with.
- Select 'Apply'.

Randomize the fragment order: This option will randomize the fragment order when the data set is created. It will not modify the appearance of the fragment list order.

- Select 'Settings'.
- Select 'Properties'.
- Press the bubble next to Random Order.
- Press 'Apply'.

Write the data set to an image: This will create an image file with the fragments. The file is placed in the same directory as Whetstone. This process may take some time.

- Add and manipulate files to achieve desired data set.
- Press 'Create'.
- Input the name of the image file you wish to create.
- Press 'Ok'.

Write the data set to USB: This will place the fragments on a USB drive. It places files on the USB in a manner that would create the desired fragmentation if the USB's file system does not optimize or manipulate the files. This will erase all of the current contents on the USB and may take considerable time.

- Add and manipulate files to achieve desired data set.

- Select ‘Settings’.
- Select ‘Properties’.
- Press the bubble next to ‘Write to USB’.
- Press ‘Apply’.
- Press ‘Create’.
- Input the location of the USB drive you wish to write to.
- Press ‘Ok’.

Initialize the random seed value: This allows users to specify how the random number generator works. It uses the python ‘random.seed’ function.

- Select ‘Settings’.
- Select ‘Properties’.
- Input the seed value in the text box next to ‘Random Seed’. (Accepts integers)
- Press ‘Apply’.

APPENDIX B

Reproducibility Log Format

This section contains the formatting for the reproducibility log.

Written to: <*output_file*>

Randomized Order: <*seed_value*>

Preferences set to: <*fragment_spacing_type*> <*cluster_size*> <*filler_byte*> <*fragment_ordering*> <*output_format*>

<*fragment_label*> moved up

<*fragment_label*> moved down

Added fragment (<*fragment_label*>)

Combined: <*fragment_labels*> into <*fragment_label*>

Renamed <*fragment_label*> to <*fragment_label*>

Removed: <*fragment_label*>

Added file: <*filename*> as fragment: <*fragment_label*>

Loaded: <*layout_input_file*>

Splitting fragment (<*fragment_label*>) from the front by size (<*split_size*>)

Splitting fragment (<*fragment_label*>) from the back by size (<*split_size*>)

Splitting fragment (<*fragment_label*>) in equal parts by size (<*split_size*>)

Splitting fragment (<*fragment_label*>) in equal <*split_frequency*> parts

Splitting fragment (<*fragment_label*>) from the front by keyword (<*split_keyword*>)

Splitting fragment (<*fragment_label*>) from the back by keyword (<*split_keyword*>)

Splitting fragment (<*fragment_label*>) in equal parts by keyword (<*split_keyword*>)