TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Raul Ezequiel Jimenez Haro 177233IVCM

FORENSIC TOOL TO STUDY AND CARVE VIRTUAL MACHINE HARD DISK FILES

Master's thesis

Supervisor: Pavel Laptev

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Raul Ezequiel Jimenez Haro 177233IVCM

KOHTUEKSPERTIISI TÖÖRIIST VIRTUAALMASINA KÕVAKETTA FAILIDE UURIMISEKS JA VÄLJAVÕTMISEKS

Magistritöö

Juhendaja: Pavel Laptev

Tallinn 2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Raul Ezequiel Jimenez Haro

13.05.2019

Abstract

Virtualization has gained rapid adoption in the last few years across all user levels, from large organizations to end-users. Digital forensic methods to analyse the virtualization platforms have not keep up with that implementation pace. This thesis focuses on hosted virtual machine hard disk sparse files. The aim is to study its structure, develop a tool to help with the understanding of how they work, and evaluate the feasibility of carving them. The research found that it is possible to leverage the structure of the files to carve them, obtain important metadata and extract data contained within the virtual disk. To achieve these results a set of experiments were designed to test the tool and knowledge about the structure of the virtual machine hard disk files. The tool helped to improve the comprehension about the files, evaluate how they can be carved and support the analysis of the files along with its forensic value.

This thesis is written in English and is 38 pages long, including 5 chapters, 26 figures and 9 tables.

List of abbreviations and terms

CLI	Command Line Interface
DFRWS	Digital Forensics Research Workshop
FBI	Federal Bureau of Investigation
GD	Grain Directory
GDE	Grain Directory Entry
GT	Grain Table
GTE	Grain Table Entry
GUI	Graphical User Interface
JPEG	Joint Photographic Experts Group
NIST	National Institute of Standards and Technology
OS	Operating System
TTU	Tallinn University of Technology
USA	United States of America
VM	Virtual Machine
VMDK	Virtual Machine Disk

Table of contents

List of figures

List of tables

1 Introduction

The advancement in information technology not only has brought new opportunities for education, entertainment, or communication to name a few, but has also enabled illicit activities to be performed with a higher sophistication level and complexity with easily available resources. At the same time, the investigative methods needed have struggled to keep up with the rhythm of this evolution.

Virtualized systems are among the technologies that have seen a rapid adoption increase in the last years, which also has enhanced its application with malicious intent. It is of big relevance the ability to retrieve and understand data from digital sources for digital investigations.

Therefore, digital forensics has become an important part of incident management in organizations and a special taskforce among law enforcement. There is a major requirement of being able to conduct a thorough investigation with the best available methods and tools to have the right level understanding of all emerging technologies.

1.1 Motivation

First, contrasting the easiness to deploy virtualization, forensic examination of such environments poses an additional layer of complexity for investigators, in which there are many unknowns originated from the different platforms and types of virtualization.

Although not a new technology, virtualization has gained much traction in the last decade. As a result, virtualization poses a new challenge for existing forensic methods and tools.

Second, file recovery and data carving are commonly focused on media files, overall there is a lack of specialization in carving other specific types of file. Common file recovery tools are not able to recover complex file structures and carving tools for specific type of files are rare. Data carving can make a crucial difference in both incident response and digital investigations.

Finally, there is a need in the digital forensic area to be able to recognize artefacts from virtual machines; furthermore, identify and recover the data from the virtual disks has become a crucial task, and a difficult one.

From these necessities arises the desire to research solutions that contributes to the improvement of the digital forensic practice.

1.2 Problem statement

Utilization of virtual machines has increased at a faster pace than the methods digital investigators have at hand to analyse them. The present research will study the structure of the Virtual Machine Disk (VMDK) files, to improve the current understanding, evaluate the feasibility of carving and study the forensic value they can provide. A tool will be developed to provide insight about the structure and contents of the files. First it will be tested to verify its contribution to the study of the file structure, then will be used against test cases designed to evaluate the feasibility of carving the files and its forensic value.

1.3 Research questions

This research aims to answer the following questions:

- What elements are needed to carve a VMDK file?
- What metadata can be obtained from the structure of a VMDK files?
- What are the advantages of doing file carving versus file recovery of VMDK files?

1.4 Scope

The scope of the research is focused on the VMDK files created by standard configuration of a hosted virtual machine (VM), which creates a set of uncompressed and unencrypted sparse files. It has three main points:

- The file format structure will be studied.
- A tool will be developed to provide a deep insight to study the files.

- It will study how the structure of the files works and how they behave in certain scenarios designed to test the feasibility of carving the VMDK files.

The developed tool is limited to find potential VMDK files and output metadata of the files identified, give insight into them and evaluate a possible carving procedure. With the output provided by the tool, a manual carving should be possible if the structure of the VMDK file is found to be fit for carving.

1.5 Outline of the thesis

Chapter 2 introduces the major concepts used for the thesis and reports a general review of existing related literature . Chapter 3 lays the theory part of the research along with the methodology employed. Chapter 4 discusses the results of the experiments performed. Chapter 5 presents a summary of the thesis, concludes achieved results and suggests future work.

2 Background

The purpose of this chapter is to give an overview of the three major concepts used for the development of the research. It begins giving a brief overview of digital forensics, followed by the definition of virtualization and file carving, ending with a short review of related existing works.

2.1 Digital forensics

With the dawn of personal computing in the late 70's the computers inevitably became criminal instruments. By the end of the next decade, the first programs that caused disruption over large networks began to appear. Moreover law enforcement identified the trend and foresaw the problems ahead that the investigators would face[1].

Those events prompted the Federal Bureau of Investigation (FBI) in the United States to create the Computer Analysis and Response Team (CART). The team identified several issues during the examinations and established a set of guidelines for processing computer evidence. This was one of the first efforts to have a group of people dedicated to computer-related evidence examination.

In 2001, the Digital Forensic Research Workshop (DFRWS) was established with the main goal of applying the scientific method to come up with solutions driven by the requirements at that moment and considering long term needs, with the input from university researchers, computer forensic examiners and analysts[2]. Further, there are many frameworks that cover the processes for digital investigation, they all share similar paths and have common goals. In particular, the DFRWS Digital Investigation Framework will be used. This framework consists of 6 major steps shown in Figure 1.

The first two steps, identification and preservation consist in recognizing the evidence, and taking the steps to preserve its state; the goal is, once the evidence has been identified, prevent any type of modification or deletion before its acquisition. The next

step, collection, is where the digital evidence is acquired, it is important to distinguish the volatility to be able to acquire it accordingly.

The examination phase applies specific tools and techniques employed to find and extract data from the digital evidence acquired, after the extraction. After the extraction, the analysis phase starts to study the data and if needed it is cross-examined it with other pieces of potentially relevant data. The last step, the presentation phase is responsible of reporting all the findings along with every step taken through all the process, the report needs to be accurate, detailed and without bias.

Figure 1. DFRWS Digital Investigation Framework.

2.2 Virtualization

Virtualization in the information technology domain is a concept that has existed since the 60's, where the mainframes needed to have solutions to share the usage of computer resources among a group of users, with the objective of increasing efficiency for both the users and the computer resources being shared [3].

The main advantage of the model is costs savings. In the last couple of years, virtualization has become a trend in the industry. It has turn out to be the best solution for data centres and organizations alike to improve the use of their resources while simplifying management and enhancing scalability.

Virtualization has transformed into a staple in modern computing, it not only has allowed organizations and data centres to lower costs, improve management and administration, to name a few, but thanks to its evolution, it has reached the average user whom can use it for example to keep running legacy systems, have a secure layer to execute untrusted software or improve debugging.

Virtualization can have different approaches when being defined, this work will use the definition from VMware will be used:

"A virtual machine is a software computer that, like a physical computer, runs an operating system and applications. The virtual machine is comprised of a set of specification and configuration files and is backed by the physical resources of a host. Every virtual machine has virtual devices that provide the same functionality as physical hardware" [4].

Complementing the definition of virtualization, we need to add a component which controls the creation and operation of the virtual machine itself, this component is called hypervisor, or virtual machine monitor. The hypervisor can be defined as:

"a process that creates and runs virtual machines (VMs). A hypervisor allows one host computer to support multiple guest VMs by virtually sharing its resources, like memory and processing"[5].

Two types of hypervisors can be distinguished, the ones that have direct access to hardware resources, also known as "bare-metal" or type 1 hypervisors, and the hosted approach which run just as another computer program inside an operating system (OS), this are called type 2 hypervisors, or "hosted"[6]. A concept visualization of hosted virtualization is shown in Figure 2.

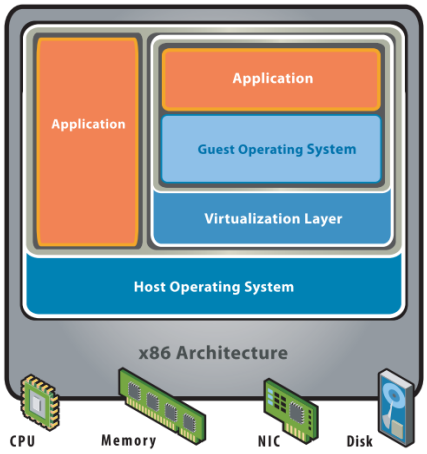

Figure 2. Hosted virtualization[6].

However, visualization can be can also be used with malicious purposes, therefore, it is also a tool used for cybercrime, it provides a good layer of obfuscation, it can be used as a disposable environment to perform illegal activities, it can serve as a covert storage, or portable environment that can be easily shared, to list a few examples, and the result of all of these is that makes it difficult for investigators to analyse and extract evidence. This is how virtualization is of particular interest for digital forensics.

2.3 File carving

File carving, as defined by Garfinkel [7] "*File carving reconstructs files based on their content, rather than using metadata that points to the content*", this means that it only looks at the raw data, based on characteristics of the file format, regardless of the filesystem metadata that may or may not be present. It is important to stress this difference with file recovery, because file recovery depends on file metadata. For digital investigations, file carving is commonly applied to unallocated space.

File carving is widely used in digital forensics as it can help discover data that previously wasn't identified, or that when applying common data recovery techniques wasn't completely recovered, for example due to anti-forensic techniques applied, intentional deletion or faulty hardware, but the data might still be in the media and could be obtained.

2.4 Related works

To the best of my knowledge, there aren't any works focusing on carving of virtual machine hard disk sparse files, however there are several studies analysing virtualization and file carving independently. Therefore, the literature relevant for this research was reviewed from two standpoints: first, works studying virtualization with a forensic approach; second, research done related to file carving.

On [7], Garfinkel presents detailed statistics about file system fragmentation showing that in modern file systems fragmentation is rare, fact that is important for this research. Additionally, this work also contributes the method of fast object validation proofing that carving can return better results when using a multi-tier method validating or discarding byte strings according to the file format being carved.

[8] proposes a structured analysis of a file format leading to a set of software requirements for a file carver. This analysis considers factors such as file format, available time, engineering capacity and data set characteristics to lead to a decision of which file carving technique to implement.

[9] makes an analysis of the existing file carving techniques used for obtaining Joint Photographic Experts Group (JPEG) files, a contribution of this work relevant for this

research is the classification of carving techniques. The study classifies different file carving techniques in file header, header-footer, file structure and metadata carving based. According to that classification, this research will have a hybrid approach, applying file header and file structure carving based techniques.

[10] gives an overview of how file carving has evolved and how FAT32 and NTFS file system work. Following, analyses the fragmentation problem and presents the wear-levelling problem, which will become more prevalent in the future with next-generation storage devices. Then continues with an in-depth examination of different file carving algorithms with its benefits and problems.

Analysis of virtual machines with a forensic perspective has been explored by [11]. This work describes the files that are created for a hosted virtual machine and establishes a scenario to study the VM where anti-forensic techniques were used. The anti-forensic technique analysed in the scenario is the use of snapshots, they mention the possibility of a deleted VM. The analysis mostly focuses on the artefacts that can be extracted from a system to identify VM activity. Nonetheless, they distinguish the need of having advanced file carving techniques for the VMDK files.

[12] present a forensic method to acquire and analyse virtual machine hard disks. Their main contribution is a comparison from two different states of the virtual hard disk to find the changes in between. Their main limitation is that a live system is used and relies on the file's metadata to identify the differences.

[13] aims to provide a point of reference for file recovery within virtual machines. They mainly present the difficulties found when trying to recover data from a virtual machine, stating that "The recovery of data from a VM is a complex task and specific, standardized, and comprehensive methods are still in the future.", so it helps to highlight the necessity of tools and methods to obtain data from virtual machines.

[14] presents a more comprehensive approach as it tries to find what evidence can be recovered from a destroyed virtual machine in the designed scenario, it outlines a method to do it, additionally it leverages information from all the virtual machine artefacts such as log files and registry entries, not only the virtual machine hard disk.

[15] suggest a procedure to investigate and a method to recover a damaged virtual machine image. Although they do cover more specific aspects of the virtual machine hard disk, their method still relies on file metadata and does not explore the carving possibilities.

The main gap I recognized in the literature is that there are no studies that analyse the possibilities of carving the VM files, let alone VMDK files specifically, this is just mentioned as a need. The works that tackle virtualization, talk about recovering artefacts in general from VMs, and when talking specifically about the virtual hard disks, they are focused in using the filesystem metadata if needed for recovery, and do not specify the type of virtual hard disk, whether it is one file or divided across several sparse files.

Most of the works about file carving focus on techniques that can be applied to the recovery of media files, while studies on general file carving centre around analysing different algorithms, all these works emphasize the problems inherent with file recovery from file system metadata alone, and highlight the importance of having a correct technique applied for each type of file.

Nevertheless, all these works highlight the importance of having an adequate knowledge of how the virtual machine disk structure works as virtualization have increased its usage for illicit activities meanwhile it has become harder for investigators to acquire evidence from these files with reliable methods and tools.

3 Theory

This chapter will present the theory on which the practical part of this research is based, detailing how the Virtual Machine Hard Disk (VMDK) files work, presenting at a high-level the algorithm for the carving helper tool and the methodology followed by the research.

3.1 Virtual Machine Hard Disk (VMDK) format

This work will investigate the files created by virtual machines (VM) under type 2 hypervisors, specifically, I will study the virtual machine hard disk (VMDK) type files, these files have the extension ".vmdk", and hereinafter will be referred to as "VMDK files".

The VMDK file format is an open format developed by VMWare and used by all major virtualization platforms, it is also used by the Open Virtualization Format (OVF) which is an open standard for packaging and distribution of virtualization solutions.[16]

The VMDK files can have a combination of the following two major characteristics: the virtual disk can be contained in one file, or it may be spread across two or more files; and the disk space for the virtual disk can be allocated at time of creation or it may start small and grow as needed. The VMDK files are also accompanied by a descriptor file, which can be embedded within the file itself or can be saved as a separate file. The descriptor file describes the layout of the data in the virtual disk[17].

When creating a VM these options can be defined, this research will study the virtual disks that grow as needed with sparse files, which are the default options when creating a VM, thus, is the most common configuration that can be found.

3.1.1 Descriptor file

The descriptor file is a flat text file formed of three main parts: header, extent description and disk data base. An example of a descriptor file can be seen in Figure 3, showing its three sections: (1) header, (2) extent description and (3) disk database.

The header gives information about the version of the format being used, the content id (CID), and the type of the disk, which indicates how the virtual disk files will be allocated, it can be a single growable virtual disk, growable virtual disk split in 2GB files, pre-allocated virtual disk or pre-allocated virtual disk split in 2GB files.

In the examples shown in Figure 3, the main information in the header section is the "twoGbMaxExtentSparse" which indicates that the virtual disk is formed by 2 or more VMDK files and that they will be allocated as the space is needed.

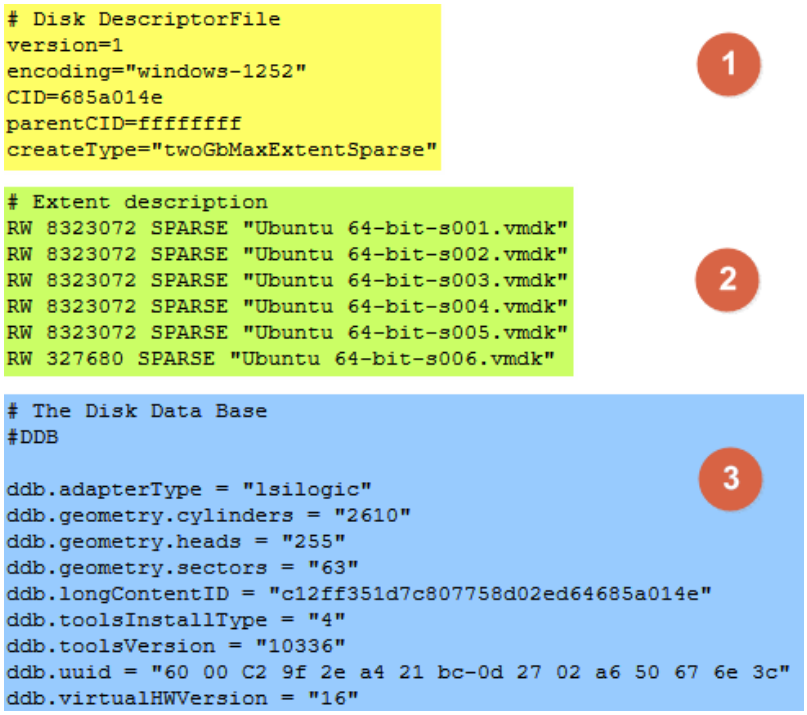

Figure 3. Descriptor file example.

In the extent description section, the file or files that form the virtual disk are listed along its access permissions, size in sectors, type and file name. The example in Figure 3 shows 6 files that can be read and written, the size of 5 files is 8323072 sectors and one file of 327680 sectors, all of them of "SPARSE" type, and their respective file names, which include a sequential indicator.

The disk data base section includes information about the geometry of the disk, such as number of cylinders, heads and sectors.

The descriptor file also has the extension ".vmdk" the same as the virtual disk, but in the case of the files that form the virtual disk the file names has a "-sXXX" suffix, where the "XXX" represents a 3-digit sequence number of the files to form the virtual disk.

3.1.2 VMDK file

The structure of the VMDK file is show in Figure 4, its main components are the header, grain directories, grain tables, and grains.

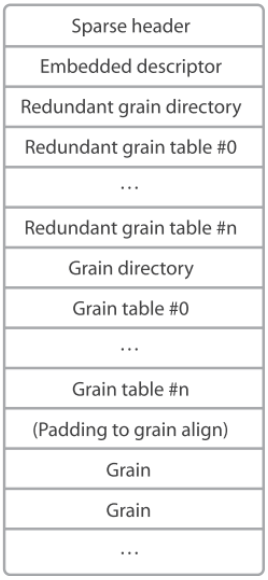

Figure 4. VMDK file structure [17].

The header is 1 sector long, which is 512 bytes, the information of the header only takes 79 bytes and its zero-padded with 449 bytes. A characterization of the header can be seen in Figure 5.

0x	00	01	02	03	04	05	06	07	08	09	A	B	C	D	E	F
00	Magic number				Version				Flags				Capacity			
10	Capacity				Grain size								Descriptor offset			
20	Descriptor offset				Descriptor size								GTEs per GT			
30	Redundant grain directory								Grain directory offset							
40	Overhead									Integrity				Compression		

Figure 5. Header structure characterization.

A complete description of each of the fields can be seen on Table 1, the value of the offset and size fields are in sectors.

The format defines two levels of metadata, a level 0 called Grain Directory (GD) and a level 1 called Grain Table (GT), each entry in the GD, called Grain Directory Entry (GDE), points to a GT. Following, each entry in the GT is called Grain Table Entry (GTE) and points to a grain, a grain is where the data itself from the virtual disk is allocated. A concept diagram is shown on Figure 6.

The file keeps two copies of the grain directories and grain tables to help protect the data in case of file corruption.

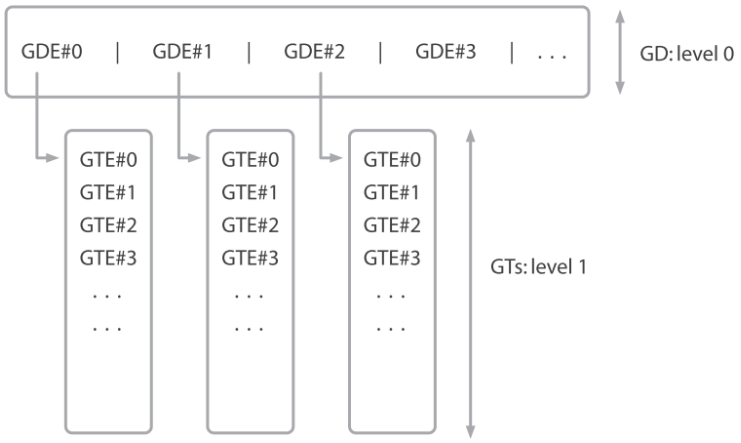

Figure 6. Metadata structure.[17]

Both the GDE and GTE are offsets which point to a GT and to a grain respectively, they are 32-bit quantities. The number of GTE in a GT is always 512, meaning that the length of each GT is 2KB. Each grain is 128 sectors long or 64KB of data. The size of the GD depends on the size of the extent.

Table 1. Header fields description [17].

Field	**Description**
Magic number.	The magic number is used to verify the validity of each sparse extent when the extent is opened.
Version.	The version number can be 1 or 2.
Flags.	Contains bits of information in the current version of the sparse format.
Capacity.	The capacity of this extent in sectors, it should be a multiple of the grain size.
Grain size.	The size of a grain in sectors. It must be a power of 2 and must be greater than 8 (4KB).
Descriptor Offset.	The offset of the embedded descriptor in the extent. It is expressed in sectors.
Descriptor Size.	Valid only if the descriptor file is embedded is non-zero.
Number of GTE's per GT.	The number of entries in a grain table. The value of this entry for virtual disks is 512.
Redundant GD offset.	Points to the redundant level 0 of metadata. It is expressed in sectors.
GD offset.	Points to the level 0 of metadata. It is expressed in sectors.

Field	Description
Overhead.	The number of sectors occupied by the metadata.
Unclean Shutdown.	Flag for consistency check.
Integrity.	Four entries are used to detect when an extent file has been corrupted by transferring it using FTP in text mode.
Compression algorithm.	Designates the algorithm to compress every grain in the virtual disk.

For example, following the sample descriptor file shown in Figure 3, the first extent named "Ubuntu 64-bit-s001.vmdk" has a size of 8,323,072 sectors, dividing the size between the grain size in sectors, we know that it can allocate a total of 65,024 grains, furthermore, dividing the number of grains between the default number of GTEs per GT we get that 127 GTs are needed to arrange them.

The representative 79 bytes of the header corresponding to that same file is shown in Figure 7 highlighting the fields that are relevant for characterizing the file.

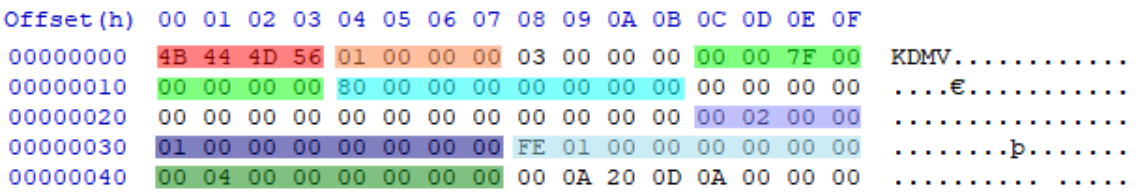

Figure 7. Header example.

The interpretation of the highlighted values is presented in Table 2 using little endian byte order.

Table 2. Header example values.

Field	Value
Header	"KDMV"
Version	01
Capacity	8,323,072 sectors
Grain size	128 sectors
Number of GTEs per GT	512
Redundant GD offset	1 sector
GD offset	510 sectors
Overhead	1,024 sectors

3.2 Carving helper tool

The tool developed for this research aims to aid the investigator in the understanding of VMDK files, the tool doesn't output a fully carved file, but provides elements to study the files, and for a manual carving procedure. When applied to unallocated space, the output of the tool identifies possible locations for a VMDK file and its contents; if applied to allocated files it helps the study of the data within the virtual disk.

The command line interface (CLI) tool was developed to explore the possibilities of VMDK file carving, the tool was build using Python programming language version 3.7.2, no external libraries were used.

3.2.1 Algorithm

The algorithm revolves around three basic functions needed to obtain the information of a VMDK file. First, a valid header needs to be correctly identified; this is the most important part, as the header contains the offsets to the metadata and data itself of the virtual disk. The second and third function is to parse the GD and GTs respectively, based on the data provided by the header. A high-level flow diagram of the overall algorithm is shown in Figure 8.

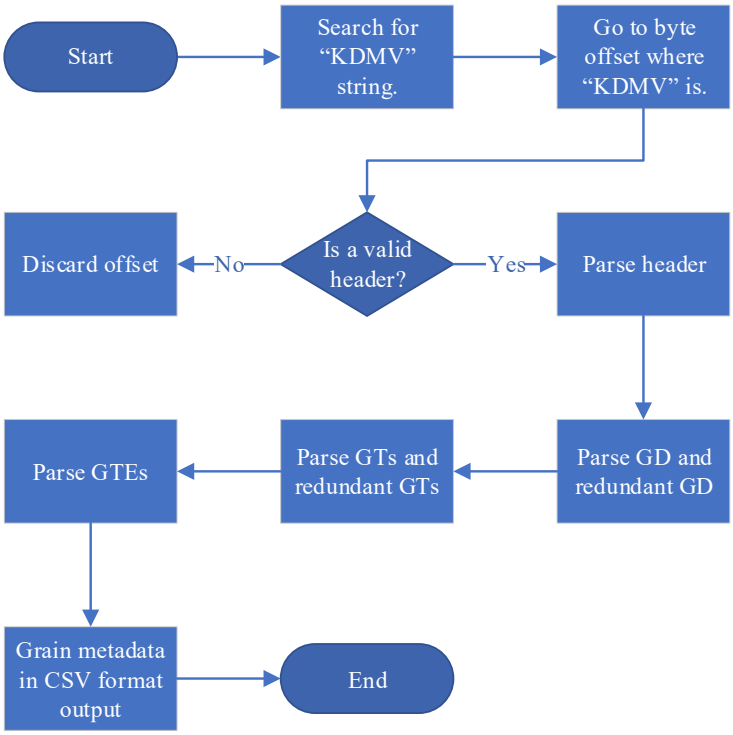

Figure 8. VMDK file carving helper tool algorithm.

The developed tool has 3 stages, in each stage the basic input is the image file, which can be from unallocated space or a whole disk image. In each stage it outputs a report file containing information hashes of the files processed, start and end time of the processing, summary of the results, and optional case information.

The first stage outputs a summary report containing the information of the headers of the VMDK files identified within the image file, and a binary file containing the header of each identified VMDK file, this binary file will serve as input for the next stages. For the second and third stage, the output is a couple of csv files per header per stage containing the metadata corresponding to the GDs, GTs and GTEs.

The initial step of the algorithm is to identify where a header might be within the disk, the algorithm searches for the bytes "\x4b\x44\x4d\x56", equal to the string "KDMV" which are the first 4 bytes of the header of the file, this is also known as the magic number, it is a signature that helps identifying the files. The byte offset where this string is found is saved.

After all the disk image has been searched and a list of possible disk headers has been created, each byte offset is explored selecting 79 bytes, which is the size of a header. Within that structure it checks for the values corresponding to the version, grain size and number of GTEs per GT fields of the header, those values are defined by the file format and should always be the same, those are the conditions that help identify a valid header.

The value of the version should be 1 or 2, the value of the grain size is 128 and the number of GTEs per GT is 512, an additional element that helps validate a valid header is the capacity field which must be a multiple of the grain size.

Once all the possible headers are checked, the tool produces a binary file which contains the 79 bytes of each valid header, at the end of it, an 8-byte number containing the byte offset of the beginning of the header and 41 zero padding is added, so each valid header is 128 byte long in the output binary file. This is to maintain an easy way to identify and loop through the known headers for the next stages.

This binary file will serve as input for the next two stages of the tool, it contains all the offsets necessary to lookup the grain directories and grain tables. The second stage

parses the GD and redundant GD and the third stage parses the GTs and its grains. The output is a pair of csv files per header for each stage.

To parse the GDs, the tool first takes the capacity of the disk and calculates how many grains, GTs and GDE the disk should have, and then it moves to each GD offset and with the previously calculated value extracts the GDEs. This process is repeated for the redundant GD. Then it parses each GTE calculating its initial and final byte offset and sector.

To parse the GTs, the tool moves to the offset of each GT, takes each GTE and calculate the coverage for each grain, then outputs the values for each allocated grain found in csv format, including header, GD, GT, and grain byte a sector offset.

In the first stage the tool receives the disk image file as input, looks for the VMDK file headers and returns as output a binary file containing the headers found, a plain text file containing a summary of the headers found and a report file of the stage.

An example of one 128-byte header from the output binary file is shown in Figure 9. The red section corresponds to the 79 bytes from the identified header as it is. The yellow section is the byte offset where the header is found within the disk image file used as input. The blue section is a 44-zero padding, which was added to facilitate the processing of the headers in the following stages.

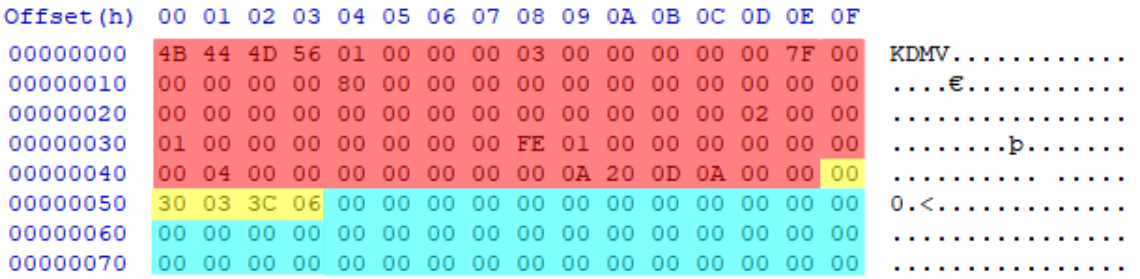

Figure 9. Header output example

The second stage of the tool processes the binary file generated at the first stage, it first uses the offset saved to look for the header. Then, for each header uses the information contained in it to look for the GD and redundant GD within the disk image file. It produces as output the stage report file and a pair of csv files containing the information of each GD including the header byte offset, GD, GT and GTE byte and sector offset.

Each file corresponds to the GD and redundant GD. Figure 10 shows an example of the csv file fields with the GD and GTs information.

Header at byte offset	GD at byte offset	GD at sector offset	GDE	GT at byte offset	GT at sector offset
26776645632	26776646144	52298137	0	26776646656	52298138
26776645632	26776646144	52298137	1	26776648704	52298142
26776645632	26776646144	52298137	2	26776650752	52298146
26776645632	26776646144	52298137	3	26776652800	52298150
26776645632	26776646144	52298137	4	26776654848	52298154

Figure 10. Second stage csv file output example.

The third stage also receives the binary file generated at the first stage along with the disk image file, using the same procedure as before, it looks for the grains and produces the stage report and a pair of csv files containing the metadata for each grain. The metadata files include the header, GD, GT and coverage, all these values are presented in its byte and sector offset, each file corresponds to the GTs and redundant GTs. Figure 11 displays the fields with the grain information.

Grain	Grain starts at byte offset	Grain starts at sector offset	Grain ends at byte offset	Grain ends at sector offset
13	Unallocated	Unallocated	Unallocated	Unallocated
14	Unallocated	Unallocated	Unallocated	Unallocated
15	Unallocated	Unallocated	Unallocated	Unallocated
16	9692229632	18930136	9692295167	18930263
17	9692295168	18930264	9692360703	18930391

Figure 11. Third stage csv file output additional columns example.

Each stage is executed independently, given the necessary inputs. The hash value of the input image file is calculated at the first stage to ensure its integrity, for following stages only the hashes of the output files are calculated.

3.3 Methodology

An experimental research will be conducted to help evaluate the feasibility of carving VMDK files, the developed tool will be tested following digital forensic principles. The first goal is to verify that the tool can correctly identify the VMDK files based on its structure. The second goal is to verify that VMDK files can be located among the unallocated space of a drive, and after the analysis of the results conclude whether the VMDK files can be successfully carved.

In the following sections, the setup for the experiments is shown, the steps of the experiments are explained and finally, the tools that will be used are briefly described.

3.3.1 Experiment design

The experimental setup will consist of two systems, designated as the analysis system and the evidence system. A pair of external storage drives to save the image file and process it, plus a set of software tools to perform the needed tasks.

The experiments to be conducted consist of four main steps as follows:

1. Prepare the evidence system as required for the test case.
2. Image the evidence system into a file.
3. Execute the file carving helper tool with the image file as input.
4. Analyse the results.

3.3.2 Experimental configuration

The purpose of having two systems is to emulate a real-life digital forensics laboratory to be able to follow forensic principles, as the research does not focus on live evidence acquisition, at least two systems are needed; one system to work as the evidence source, and one system to function as the laboratory system.

The evidence system is where the test configurations will be created according to the test cases and will be imaged into a disk image file to be processed. The analysis system is where the carving helper tool will be executed, and the output will be analysed.

Both systems have similar technical specifications, however, the analysis system has more capabilities on processing power and memory capacity, which can help to speed up the processing times of the evidence acquired.

The aim of the evidence system is to represent the most common type of system which could be encountered on real life digital investigations; therefore, its specifications can be considered as standard, including the default OS installation without any customization.

For the research purposes, having these two systems also facilitate the workflow, focusing the processing and analysis work on one system, and having other system

available to create the different scenarios without the need of having to swap different hard disk drives.

The specifications of each of the experiment elements are showed in Table 3.

Table 3. Experiment elements specifications.

Element	**Specifications**
Evidence system	Fujitsu Lifebook S761 Processor: Intel(R)Core(TM) i5-2520M CPU @ 2.50GHz, 2501 MHz, 2 Core(s), 4 Logical Processor(s) Installed Physical Memory (RAM): 10.0 GB
Analysis system	Lenovo ThinkPad X230 Processor: Intel(R)Core(TM) i7-3520M CPU @ 2.90GHz, 2901 MHz, 2 Core(s), 4 Logical Processor(s) Installed Physical Memory (RAM): 16.0 GB
Evidence drive	Model: HGST HTS545025A7E380 Disk Size: 232.33 GB (249,464,614,912 bytes) Volume Serial Number: 806686FC
External drive #1	Model: Seagate Expansion Desk SCSI Disk Device Volume Name: Seagate Expansion Drive Size: 4.55 TB (5,000,845,586,432 bytes) Volume Serial Number: A4D0BF9A
External drive #2	Model: WD My Passport 0741 USB Device Volume Name: My Passport Size: 931.48 GB (1,000,169,533,440 bytes) Volume Serial Number: 1AEA6007

The evidence drive is installed in the evidence system, it will not be extracted for the imaging, a software write blocker will be used to maintain the integrity of the data. The external drive #1 is where the image from the evidence system will be saved at the moment of creation. A copy of this image will be made to the external drive #2 which the analysis system will use for processing. Preserving the original image file in the external drive #1 has the purpose of preserving the original data intact to adhere to forensic principles procedure.

3.3.3 Test cases

To evaluate the file carving helper tool, a set of test cases was designed to be certain that the tool performs as expected and that the output can be useful for a digital investigation. Each test case has an expected result and a validation procedure. A summary of the test cases can be seen in Table 4.

Table 4. Test cases.

Test case	Description	Expected result
TC – 01	System clean.	No VMDK files should be found.
TC – 02	System installed with one clean VM.	VMDK files of the VM should be found.
TC – 03	System with deleted VM.	VMDK files of the VM should be found.
TC – 04	System with deleted VM and a file.	Locate file using the information provided by the tool.

In the test case TC – 01 the evidence system is a system freshly installed with Microsoft Windows 10 OS with default installation, no additional software is installed; however, a reference data set was added for the drive to have some content and help testing the accuracy of the tool. The disk image generated for this test case will also serve as a control image.

The reference data set is comprised of forensic images available by NIST to test software applications with file carving capabilities[18], [19], these images contain graphic, video, document, archive, and audio files. The full set of files used for this research is shown on Table 5.

Table 5. Reference data set files.

File	SHA-1
graphic-src.zip	26B6641FC41834ACFBDD9BC5EC4A8C57033CFF71
graphic-image-files.zip	D2F703CE537E0632414F907E08C65EED67A3F17F
video-src.zip	A93E5ACA9CB55BD7116921E1A2CF8E69A0F4392A
L0_Documents.dd	4767425B6BF0A9D21089546FBEC938AED401B8A1
L0_Archive.dd	83D14584277DF44667A7E2242F1744A62F60B3B3
L0_Audio.dd	3BC5375227E48E489F48AFA5C9D39DCBC9DA308D
images.zip	F009E84B8F27B22BDE4D593B85A50946FE1C9603

The results from the TC-01 will be validated by comparing the results of a string search on the image file, the search for “KDMV” string might return some results. The expected result is that the carving helper tool will not find any indication of a VMDK file.

In test case TC – 02, using the previously installed system for TC-01, VMWare Player will be installed, this is the type 2 hypervisor. Next, a VM will be installed on it; the goal is that the tool identifies the VMDK files created by the VM in evidence system.

The VM to be installed will use the default configuration provided by the hypervisor, the OS to be installed is Ubuntu 18.10, the latest version, and the default virtual disk created by it is a 20 GB disk, which is divided in 6 VMDK sparse files, these are the files that the tool needs to identify.

The expected result is that the tool can identify all the VMDK files created by the VM. To validate the TC-02, before the imaging, the byte offset location of the 6 VMDK files will be logged, these offsets should match the output files of the carving helper tool.

In test case TC – 03 the VM created for TC – 02 will be deleted from the hypervisor, the unallocated space will then be imaged, the goal is that the tool identifies the VMDK within the unallocated space of the evidence system. To help validate this result, a data recovery tool will be used on the unallocated space, so that a result can be compared with the output from the carving helper tool. This will help compare how a data recovery works versus a possible carving.

For test case TC – 04 the evidence system will be returned to a known state using the image from TC – 01, then the same procedure for TC-02 will be followed to have a VM installed on the system, additionally a plain text file will be saved in the VM disk, and the file will be located to know its byte offset within the virtual disk.

The content of the plain text file is “This is my evidence file. REJH” which is just a simple unique string that will allow me to locate the file easily with a search in a hex editor to log the byte offset location within the virtual disk. The purpose of the file is to be able to have a known byte offset location of a file within the virtual disk to locate it after using the output from the carving helper tool.

The evidence system will be imaged, and the carving helper tool will be executed on each sparse file of the VM prior to is deletion; following, the VM will be deleted from the evidence system and the unallocated space imaged.

Using the file carving helper tool output, I should be able to map the location of the file within the virtual disk to an exact grain in a VMDK file. First, the carving helper tool will be executed on the image prior to deletion, this should confirm that the output is helping in finding the actual file created within the virtual disk. This will help validate the next step.

Finally, knowing that the output does help to map the location, the carving helper tool will be executed on the unallocated space after the VM deletion and using the results I should be able to locate the same file among the sparse files identified on the unallocated space.

3.3.4 Software tools

To conduct the experiments adhering to a forensic procedure, software that complies with the same principles is needed. I need software that assures the integrity of the data, in this case, I need to make sure that the image file is the same as the evidence drive, and through the process of analysis I need to maintain that certainty.

Paladin bootable Linux distribution will be used, this distribution includes the tools with the needed characteristics. It is a distribution that provides a forensic environment to perform several tasks, it has passed tests by the National Institute of Standards and Technology (NIST) which provide us with a certain level of confidence that the environment will perform as expected.

It will be used for imaging, including hashing and logging of the process. For imaging, the distribution provides a software-write blocker, it achieves this by disabling auto mounting the connected drives, and mounting in read-only mode the drive to image, this helps assuring that the evidence drive will not be modified and will be imaged untampered.

The distribution integrates hashing during the process of imaging, it calculates the hash of the drive as it is being imaged, at the end hashes the generated image. Then, the hashes can be compared to verify that the output file is the same as the data on the

evidence drive. Every command executed for the process of imaging is logged along of the hashing process.

The type 2 hypervisor to be used is VMWare Player version 15, from the developers of the VMDK file format. According to each test case specification, a virtual machine will be created using default configuration.

At the moment, there isn't any tool that specializes in carving VMDK files, most of the carving tools available focus on media files, nevertheless there are tools that while not specialized in VMDK files, they are able to attempt it. To aid in the validation of the results, PhotoRec will be used, which is a data recovery tool that has passed forensic file carving tests from NIST [20], and is capable of carving 480 file extensions based on their signature [21], without the need of further customization of the tool.

Additionally, the research "The analysis of file carving process using Photorec and Foremost"[22] have proofed that PhotoRec overall is a better tool when compared with Foremost. More specific, PhotoRec is faster and can return a larger number of valid files.

Other tools that will be used are a hex editor, HxD editor will be used in Windows environment in the analysis system, and WxHexeditor will be used in Linux environment both in the analysis system and in the VM created in the evidence system. The hex editor helps to visualize the raw data from files and drives, inside I can locate the position within the file or drive using the byte or sector offsets.

Access Data FTK imager will be used in the analysis system to load the images, this tool enables to explore the image or file loaded in a hexadecimal view, it also interprets the filesystem which allows to navigate within the file tree structure.

This chapter presented the workflow and configuration for the experiments. It also detailed the test cases to be executed and the tools to be used. The following chapter will present the results and discuss the findings.

4 Results and analysis

This chapter will present a detailed analysis of the results obtained for each test case, along with additional tasks performed to extend the study of the VMDK files.

4.1 Test case TC – 01

This test case was designed to verify that the tool could successfully identify where a header is and distinguish it from any other "KDMV" string which in any system could be found several times.

The output of the tool (Figure 12) showed that there were no headers found, which matches the expected result. The full output report can be seen on Appendix 2 – TC – 01 report.

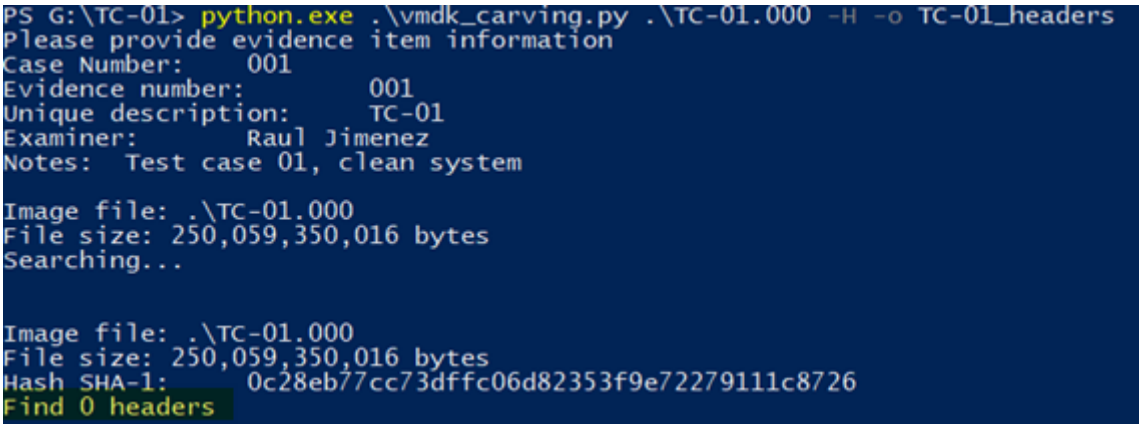

Figure 12. VMDK carving tool output for TC-01.

To validate the result the image file of the evidence system was loaded into a hex editor and a search for "KDMV" was performed, the search returned 37 results, this means that inside the disk image the string "KDMV" is found at 37 locations. The results from the hex editor are shown on Figure 13.

This result confirms that the tool is performing as expected, it is successfully distinguishing a common existing string from a string belonging to a VMDK header. It also gives an initial certainty that the tool will be able to identify valid VMDK files starting from its headers.

The next test case will help to verify that the tool is able to recognize valid headers in which not only the magic number is searched, but additional parameters are validated to confirm that the section starting corresponds to a possible VMDK file.

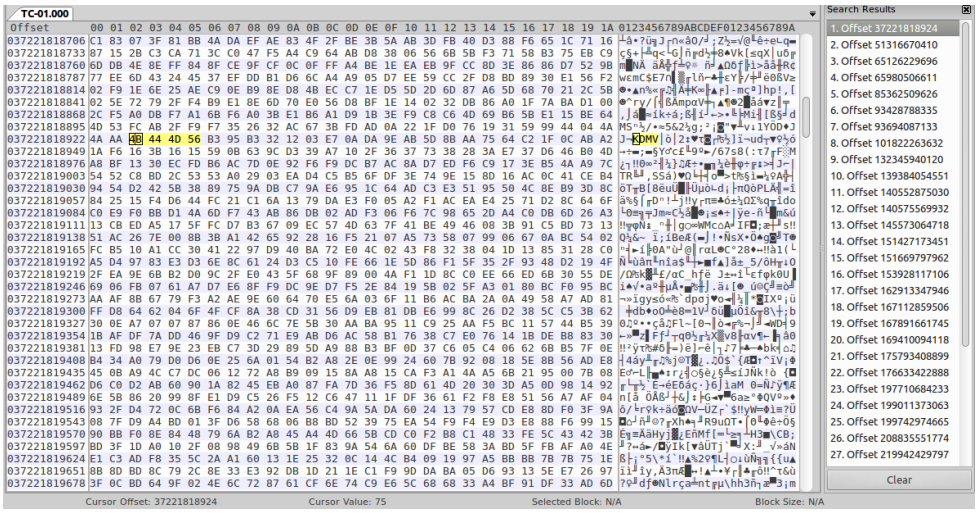

Figure 13. Search results for "KDMV" from the hex editor.

4.2 Test case TC – 02

For TC-02 the installation of the VM on the evidence system resulted in 6 VMDK files, these files are accompanied by the descriptor file which is not embedded and describe these 6 files. As seen on Figure 14 where: (1) is the descriptor file, its contents, (2) is the description of each sparse disk that build the whole virtual disk, including the capacity, from the 6 files, 5 files are 8,323,072 sectors long and 1 file 327,680 sectors long; (3) the files described are shown in the file tree.

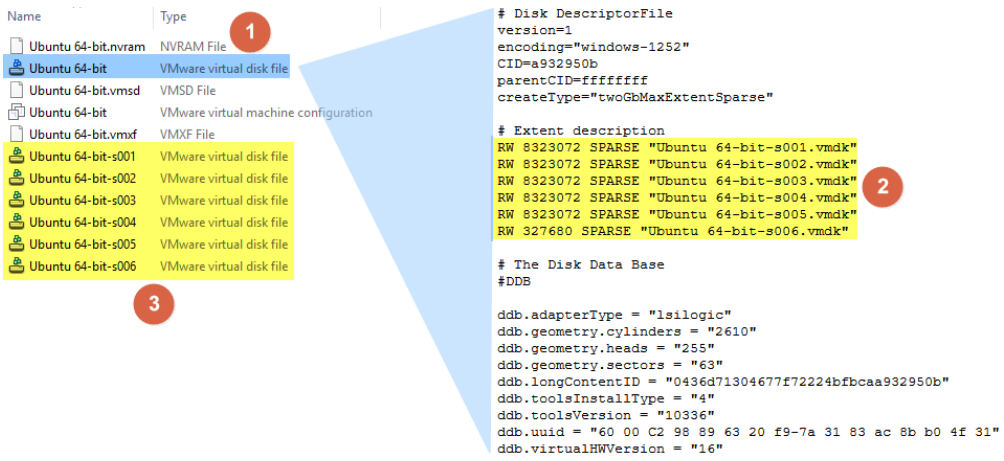

Figure 14. VM files on evidence system.

Using FTK imager it is possible to locate each file to know its position in the disk, this will be compared with the output of the tool. The evidence drive is loaded on FTK imager, navigating in the file tree to the location of the VM files, and selecting each file will show at the bottom status bar the location of the file, to compare with the carving tool.

The analysis on FTK imager is shown in Figure 15, (1) show the evidence drive loaded; (2) shows the file located; (3) shows the positions of that file, I will be logging the physical sector offset for each file.

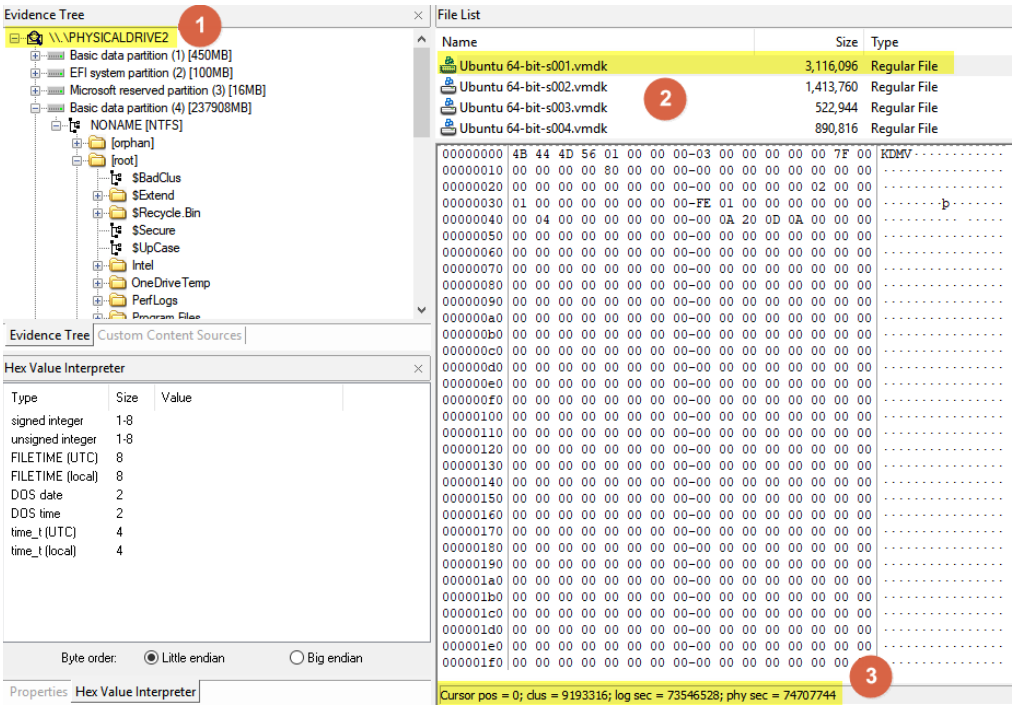

Figure 15. Evidence drive on FTK Imager.

The sector offsets for each file is shown on Table 6. These offsets will be used to validate the output from the carving tool.

Table 6. Files sector offset location on evidence drive.

File	**Sector offset**
Ubuntu 64-bit-s001.vmdk	74,707,744
Ubuntu 64-bit-s002.vmdk	79,612,224
Ubuntu 64-bit-s003.vmdk	82,276,632
Ubuntu 64-bit-s004.vmdk	85,954,896
Ubuntu 64-bit-s005.vmdk	83,674,296
Ubuntu 64-bit-s006.vmdk	6,191,608

The tool output shows that 6 headers were found, and two files were created, the summary file of the headers and the binary file containing the headers information for the next stage, this is shown on Figure 16.

```
Image file: TC-02.000
File size: 250,059,350,016 bytes
Hash SHA-1:     0397b44940b07ebc4cf5f8e7aa5e4f3a48dae912
Find 6 headers
Output files:
        TC02_headers.txt with hash SHA-1 7de07c6c981647292d6522e631b898ace90e3384
        TC02_headers.bin with hash SHA-1 197faf00e2d7d5ed7ccaca149f39f729a011160b
```

Figure 16. Carving tool output for header search.

The header summary file (TC02_headers.txt) gives the byte and sector offset, additionally it outputs the main fields from each found header, on Figure 17 the information related to the first header found is displayed.

```
Header is at byte offset:       3170103296
Header is at sector offset:     6191608
Magic number:   1447904331
Version:        1
Capacity:       327680
Grain size:     128
Number of GTEs per GT:  512
Redundant grain directory offset:       1
Grain directory offset: 22
Overhead:       128
```

Figure 17. Header summary output file example.

Using the information from the output file TC02_headers.txt generated by the carving tool, I can extract the offsets of the headers found and compare them to the ones obtained manually from the evidence drive. Table 7 shows the information extracted from the headers summary file generated by the carving tool.

Table 7. Headers offsets found by the carving tool.

Header	**Capacity(sectors)**	**Sector offset**	**Byte offset**
Header 1	327,680	6,191,608	3,170,103,296
Header 2	8,323,072	74,707,744	38,250,364,928
Header 3	8,323,072	79,612,224	40,761,458,688
Header 4	8,323,072	82,276,632	42,125,635,584
Header 5	8,323,072	83,674,296	42,841,239,552
Header 6	8,323,072	85,954,896	44,008,906,752

The full header summary output file for TC-02 can be seen on Appendix 3 – TC – 02 header summary file.

The comparison of the 6 headers found by the tool matches the offsets of the 6 files generated by the VM, this shows that the tool is correctly identifying 6 existing VMDK files, and its known location within the disk.

At this point I cannot identify which found header corresponds to which file, and the outputs shows that files are not allocated in order, meaning that to order the files I cannot consider the offset at which they are located.

Continuing with the first header as example, with the information provided at this point by the tool I can only identify the smallest file, which is 327,680 sectors long, I can validate this information by looking into the evidence drive and look for the smallest sparse file created by the VM. On Figure 18 a comparison between the data shown by FTK imager (1) and with the output file TC02_headers.bin (2) is shown, the header data matches.

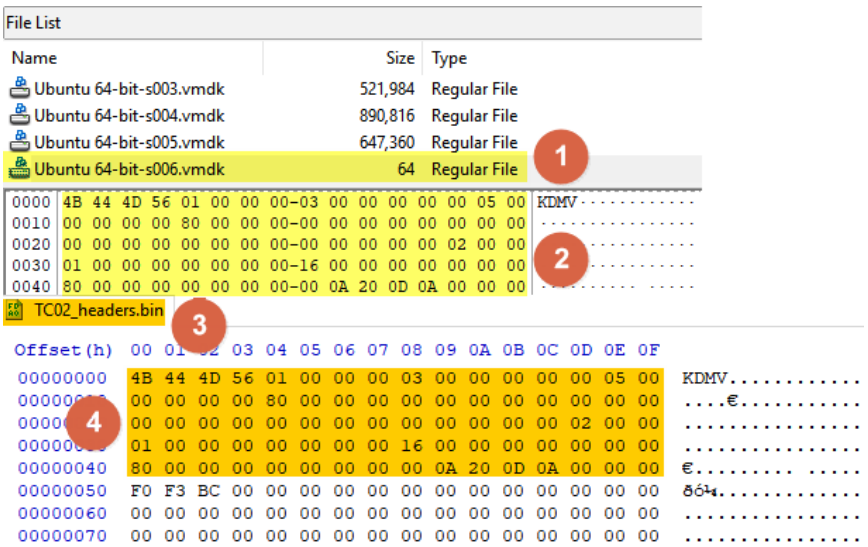

Figure 18. Header comparison.

With this successful verification, I can validate that the carving tool is identifying and saving the valid headers information for further processing.

So far, I have proofed that the tool is correctly identifying where a valid header is, discerning it from strings that could've been considered as valid, also, the headers found as valid correspond to an actual valid file and the information being generated by the tool matches the files created by the VM.

At this point, the main identified challenge is distinguishing the different files, the tool is correctly identifying where there is a header, thus marking where there is a VMDK file, but using only the information obtained there is no way to distinguish the order of the sparse files, meaning that the files potentially can be carved but cannot be arranged without additional input.

However, analysing the information further, there is a way in which at least the first and last sparse files can be identified, the last disk is the sparse file with the least capacity, and the first sparse file should have the boot sector in it.

The VM at the boot process looks for the boot sector on the first sparse file, in this case Ubuntu 64-bit-s001.vmdk. In this disk the grain 0 should correspond to the first 512 bytes on the virtual disk.

This analysis can be performed using the output from third stage which shows the location of the grains and comparing it to the output from the actual boot sector from the virtual disk on the VM. The search can be done by looking for the grain 0 on all GDEs 0, which means to search the first grain of the first grain table on each identified file, and one of these offsets should point to where the boot sector of the virtual disk is.

First, I extract the boot sector from the virtual disk in the VM to compare it after. Figure 19 shows the hexadecimal dump for the first 512 bytes of the disk.

```
user@ubuntu:~$ sudo hexdump -n 512 -C /dev/sda
[sudo] password for user:
00000000  eb 63 90 10 8e d0 bc 00  b0 b8 00 00 8e d8 8e c0  |.c..............|
00000010  fb be 00 7c bf 00 06 b9  00 02 f3 a4 ea 21 06 00  |...|.........!..|
00000020  00 be be 07 38 04 75 0b  83 c6 10 81 fe fe 07 75  |....8.u........u|
00000030  f3 eb 16 b4 02 b0 01 bb  00 7c b2 80 8a 74 01 8b  |.........|...t..|
00000040  4c 02 cd 13 ea 00 7c 00  00 eb fe 00 00 00 00 00  |L.....|.........|
00000050  00 00 00 00 00 00 00 00  00 00 00 80 01 00 00 00  |................|
00000060  00 00 00 00 ff fa 90 90  f6 c2 80 74 05 f6 c2 70  |...........t...p|
00000070  74 02 b2 80 ea 79 7c 00  00 31 c0 8e d8 8e d0 bc  |t....y|..1......|
00000080  00 20 fb a0 64 7c 3c ff  74 02 88 c2 52 bb 17 04  |. ..d|<.t...R...|
00000090  f6 07 03 74 06 be 88 7d  e8 17 01 be 05 7c b4 41  |...t...}.....|.A|
000000a0  bb aa 55 cd 13 5a 52 72  3d 81 fb 55 aa 75 37 83  |..U..ZRr=..U.u7.|
000000b0  e1 01 74 32 31 c0 89 44  04 40 88 44 ff 89 44 02  |..t21..D.@.D..D.|
000000c0  c7 04 10 00 66 8b 1e 5c  7c 66 89 5c 08 66 8b 1e  |....f..\|f.\.f..|
000000d0  60 7c 66 89 5c 0c c7 44  06 00 70 b4 42 cd 13 72  |`|f.\..D..p.B..r|
000000e0  05 bb 00 70 eb 76 b4 08  cd 13 73 0d 5a 84 d2 0f  |...p.v....s.Z...|
000000f0  83 d0 00 be 93 7d e9 82  00 66 0f b6 c6 88 64 ff  |.....}...f....d.|
00000100  40 66 89 44 04 0f b6 d1  c1 e2 02 88 e8 88 f4 40  |@f.D...........@|
00000110  89 44 08 0f b6 c2 c0 e8  02 66 89 04 66 a1 60 7c  |.D.......f..f.`||
00000120  66 09 c0 75 4e 66 a1 5c  7c 66 31 d2 66 f7 34 88  |f..uNf.\|f1.f.4.|
00000130  d1 31 d2 66 f7 74 04 3b  44 08 7d 37 fe c1 88 c5  |.1.f.t.;D.}7....|
00000140  30 c0 c1 e8 02 08 c1 88  d0 5a 88 c6 bb 00 70 8e  |0........Z....p.|
00000150  c3 31 db b8 01 02 cd 13  72 1e 8c c3 60 1e b9 00  |.1......r...`...|
00000160  01 8e db 31 f6 bf 00 80  8e c6 fc f3 a5 1f 61 ff  |...1..........a.|
00000170  26 5a 7c be 8e 7d eb 03  be 9d 7d e8 34 00 be a2  |&Z|..}....}.4...|
00000180  7d e8 2e 00 cd 18 eb fe  47 52 55 42 20 00 47 65  |}.......GRUB .Ge|
00000190  6f 6d 00 48 61 72 64 20  44 69 73 6b 00 52 65 61  |om.Hard Disk.Rea|
000001a0  64 00 20 45 72 72 6f 72  0d 0a 00 bb 01 00 b4 0e  |d. Error........|
000001b0  cd 10 ac 3c 00 75 f4 c3  e9 0c 75 f2 00 00 80 20  |...<.u....u.... |
000001c0  21 00 83 fe ff ff 00 08  00 00 00 f0 7f 02 00 00  |!...............|
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200
```

Figure 19. Boot sector from virtual disk.

The boot sector from the virtual disk corresponding to the first sector, is the same data I should find on a grain 0 within the files located by the carving tool.

The next step is to find the grain 0 of all the files found by the tool, to do this, I will use the files generated in the third stage of the tool, which parses the GTEs and gives me the location of each grain.

The carving helper tool outputs a csv file for each GD, therefore there are 2 csv files per header, one for the redundant grain directory and one for the grain directory. The information I'll be looking is highlighted in Figure 20.

GT at sector offset	Grain	Grain starts at byte offset	Grain starts at sector offset	Grain ends at byte offset	Grain ends at sector offset
6191610	0	Unallocated	Unallocated	Unallocated	Unallocated
6191610	1	Unallocated	Unallocated	Unallocated	Unallocated
6191610	2	Unallocated	Unallocated	Unallocated	Unallocated
6191610	3	Unallocated	Unallocated	Unallocated	Unallocated
6191610	4	Unallocated	Unallocated	Unallocated	Unallocated

Figure 20. Grain metadata example.

Extracting all the offsets for grain 0 from the output files, Table 8 is created, as shown, there are only two possibilities, as the other ones are unallocated, which means no data is stored on those grains.

Table 8. Grain 0 offsets.

Header	**Byte offset of grain 0 beginning**
Header #1	Unallocated
Header #2	38,250,889,216
Header #3	41,465,184,256
Header #4	Unallocated
Header #5	Unallocated
Header #6	Unallocated

Looking into the possible options, the boot sector is found on the header #2, show in Figure 21. The sector found with the information given by the carving helper tool matches the boot sector within the VM, the last step is to verify that this sector is found on the first sparse file Ubuntu 64-bit-s001.vmdk.

The verification can be seen on Figure 21, where (1) is the output sector on the evidence drive, and (2) is the sector on the image file, and both matches the previously sector extracted from the virtual disk within the virtual machine.

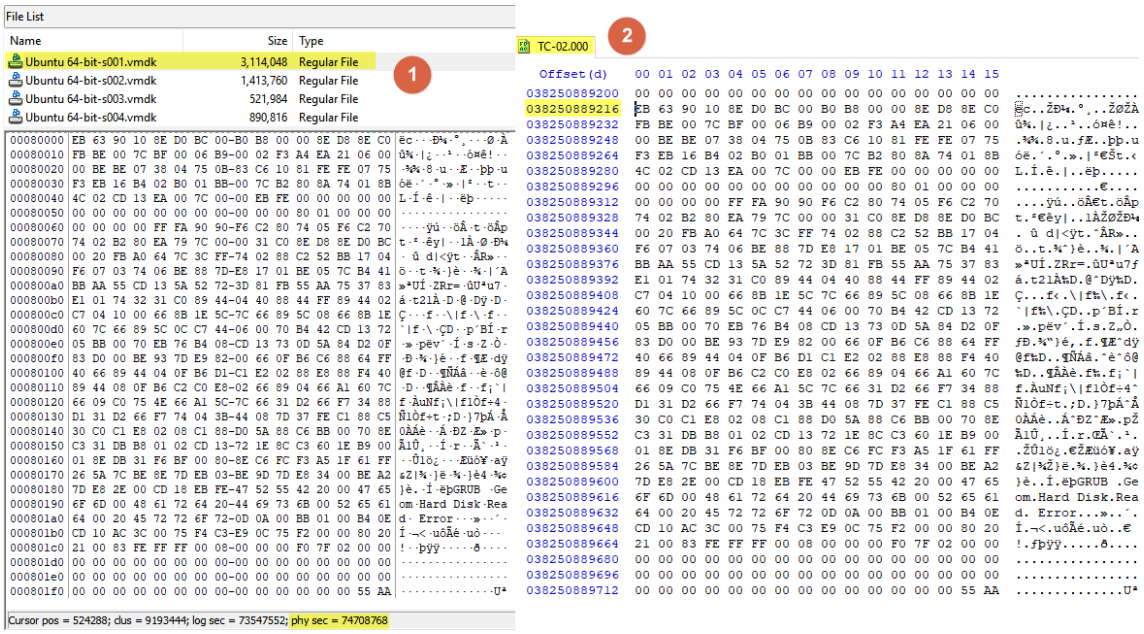

Figure 21. Boot sector on evidence drive image file.

By looking at the summary of the headers, of the six headers identified, five of them have the same size and one is of lower capacity, this matches with the information on the descriptor file, the header with the least capacity should correspond to the file Ubuntu 64-bit-s006.vmdk, in this case, header #1 shown by the summary report.

The comparison between the headers of the files from the evidence drive and with the image file using the offsets found is shown in Figure 22.

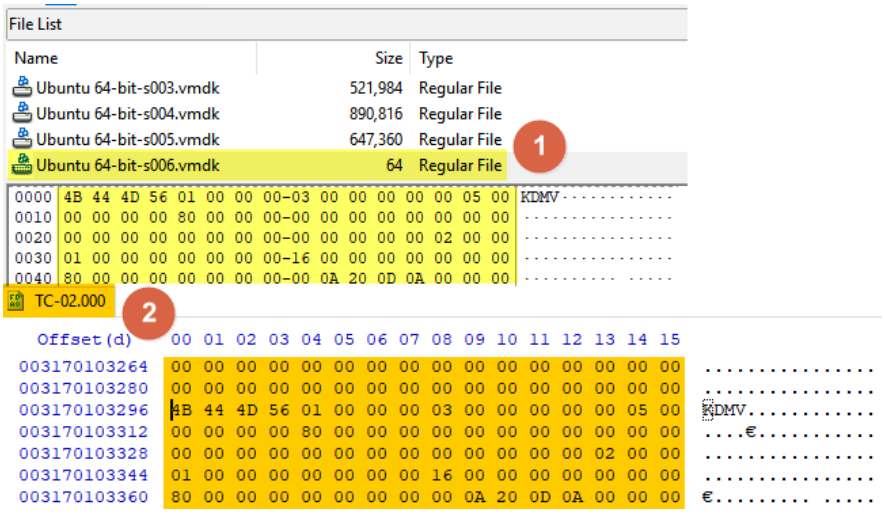

Figure 22. Headers comparison between evidence drive and image file.

This analysis can help narrowing the search for the correct arrangement of the files, but with external information this can be completed. In this analysis I am just considering the output of the carving tool, however if the investigator also has additional information, such as the descriptor file, or filesystem metadata, then the order has more possibilities of being found.

In the next test cases I will evaluate if the tool can locate the VMDK files among the unallocated space and what information could be useful for a forensic investigation.

4.3 Test case TC – 03

For the TC-03 the VM created for the previous case was deleted directly from the hypervisor, which causes the files from the VM to be deleted without passing by the recycling bin.

The carving tool returned 5 found headers on the unallocated space, which without looking further, is one file less than it should have recognized. The full header summary from the carving tool can be seen on Appendix 4 – TC – 03 header summary file.

The file recovery tool used was Photorec, and it returned 4 found files (Figure 23), the headers from these files match with the files found by the carving tool

```
<fileobject>
  <filename>f73546528.vmdk</filename>
  <filesize>2511093760</filesize>
  <byte_runs>
    <byte_run offset='0' img_offset='38250364928' len='2511093760'/>
  </byte_runs>
</fileobject>
<fileobject>
  <filename>f78451008.vmdk</filename>
  <filesize>1364176896</filesize>
  <byte_runs>
    <byte_run offset='0' img_offset='40761458688' len='1364176896'/>
  </byte_runs>
</fileobject>
<fileobject>
  <filename>f81115416.vmdk</filename>
  <filesize>715603968</filesize>
  <byte_runs>
    <byte_run offset='0' img_offset='42125635584' len='715603968'/>
  </byte_runs>
</fileobject>
<fileobject>
  <filename>f82513080.vmdk</filename>
  <filesize>1167667200</filesize>
  <byte_runs>
    <byte_run offset='0' img_offset='42841239552' len='1167667200'/>
  </byte_runs>
</fileobject>
```

Figure 23. Files found by Photorec.

In a common file deletion, where the file goes to the recycle bin, according to [10] the cluster links are preserved and the cluster where the file was found is just marked as available, this facilitates the recovery if needed.

The file that was not identified by the carving helper tool was the smallest sparse file, it is uncertain how the deletion of the files is being done by the hypervisor, one possibility could be that it attempts to delete the header, which is why this file was not found, while some data related to it might still be present. However, if this would be the case, neither file would likely to be found.

Another possibility is that the geometry of the disk is having some effect and somehow the header could still be present within the allocated space. Although further testing is needed to study how the deletion from the hypervisor is working, the files identified by the carving helper tool hold most of the VM information, which for the purposes of an investigation could be sufficient enough.

On the other hand, in the files recovered by Photorec, it recovered a file larger than the size of the virtual disk, which is incorrect, it is identifying a file of 96 GB, while the whole virtual disk is 20 GB, and the maximum size for each sparse file is 4 GB.

Analysing the results from the file recovery tool it seems as it correctly identifies the header of the files, to recover the files relies on file system data, such as cluster size, in this case it is recovering more data than it should as it is not reading the metadata of the VMDK file.

Judging by the results, filesystem-based recovery can be less accurate than a manual carving attempt using the helper tool, as it only considers the structure of the file and moves within those limitations and it can provide a more exact result, additionally the carving tool identified one additional file to the recovery tool.

The next test case will try to evaluate how useful is the information provided by the carving tool to find data within the identified files in the unallocated space.

4.4 Test case TC – 04

For TC-04 I am trying to verify if a known location within the virtual disk can be located within the unallocated space using the output generated by the tool.

Within the VM the plain text file was created to be easily identified in a search with a hex editor, the file is located at the bye offset 6,754,312,200 in the virtual disk (Figure 24). The goal is to map this location to a grain and check if that grain is recoverable from the unallocated space.

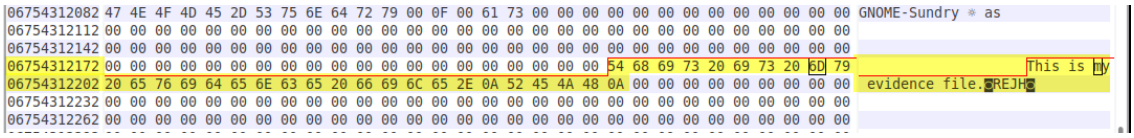

Figure 24. Text file inside the VM.

To conduct the analysis, I need to state some facts about the geometry of the sparse files. Each sparse file has a maximum size of 8,323,072 sectors, which are organized in 127 GDEs which in turn are 127 GTs. Each GT contains 512 grains and each grain is 128 sectors long. Each grain is 65,536 bytes long and consequently each GT is 33,554,432 bytes long.

To obtain the exact location within the image the following equations are needed:

- Equation (1) is and integer division between the known location (X) and the GT coverage, in this case the known location is the offset of the text file. This result is the GT to look for.

6,754,312,200 // 33,554,432 = 201 (1)

- Equation (2) is the modulo operation between the known location (X) and the coverage of each GT. Equation (3) is an integer division between the previous result and the coverage of each grain. This result is the grain to look for.

6,754,312,200 mod 33,554,432 = 9871368 (2)

9,871,368 // 65,536 = 150 (3)

- Equation (4) is the modulo operation between the known location (X) and the grain coverage to know the exact position in the grain. This is the offset within the grain.

6,754,312,200 mod 65,536 = 40,968 (4)

According to these results, the file should be at offset 40,968 in the grain 150 on GT 201. As previously indicated, each file has a maximum of 127 GTs, so an extra modulo operation is needed to obtain that the GT to look for, to which the result is 74 indicating the GDE to look for.

Executing the carving tool on the evidence drive before the files are deleted helps me confirm that the values found are correct to locate the file, as show in Figure 25.

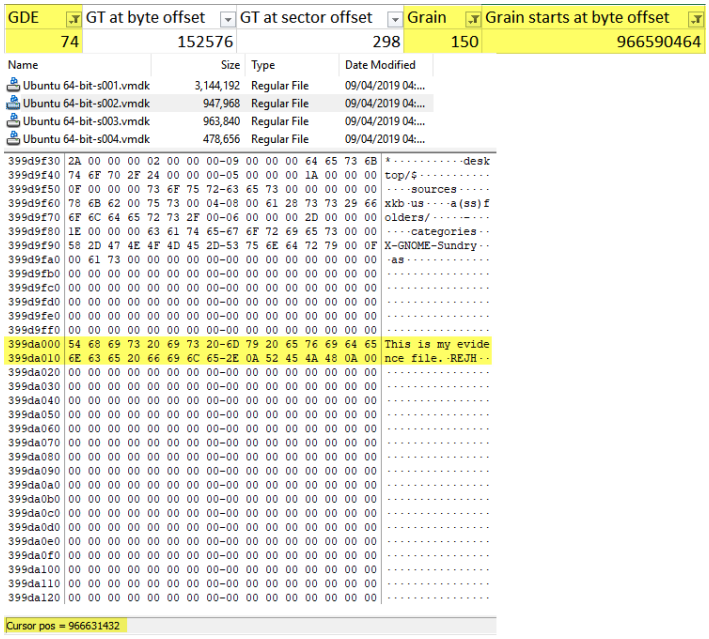

Figure 25. File found on evidence drive.

The next step is to investigate the output files from the carving tool executed on the unallocated space of TC-04, in which although it didn't identify the 6 files, the data might be within the 5 found files. The compiled offsets are shown in Table 9.

Table 9. Offsets in evidence image file.

Header	**Start offset for grain 150 in GT 74**
Header #1	9,734,828,032
Header #2	10,913,030,144
Header #3	12,841,140,224
Header #4	13,428,867,072
Header #5	18,461,372,416

Testing all the possible values, the file is found on the space related to header #3, as shown on Figure 26.

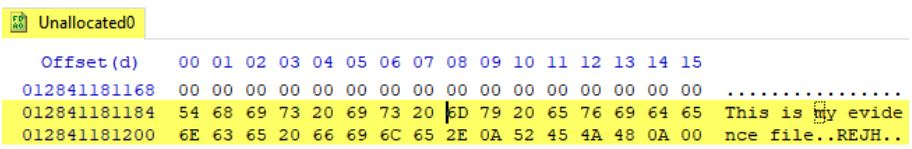

Figure 26. File located among the unallocated space.

This confirms that data can be found within the VM files from unallocated space, although not all files are identified, still valid data can be extracted. Additionally, it validates the output from the tool can be used to have a deep insight about the VMDK files, not only for carving purposes on unallocated space, but to analyse allocated VMDK files. Moreover, it shows that the data generated by the tool can be used to map known locations inside the virtual disk and an image file with the VMDK sparse files.

For data within unallocated space it can help to perform a manual carving if enough data can be found, or it can uncover previously unknow metadata about the files, which in a digital investigation every bit of data obtained can be helpful. It can point to where is more likely to found data belonging to the disk, by providing start and end offsets, using this information an investigator could potentially do a manual carving of the VMDK sparse files or at least identify data found within.

5 Conclusions

Digital forensics has a big challenge keeping up with emerging and evolving technologies, virtualization is one of these technologies that needs a new set of tools and methods to analyse them. File carving is a technique to extract files from raw data based on its structure; it is commonly used in media files. However, its application to other type of files can result in valuable evidence being discovered.

This thesis aimed to study the structure of virtual machine disk (VMDK) files, to improve the understanding about them, develop a tool to aid in the analysis, evaluate the feasibility of a carving operation and assist in such procedure.

The research proofed that the structure of VMDK files can be used to carve them, it has a two-layer structure. To locate these two layers, the header of the file is the basic piece of data that needs to be obtained to be able to perform the carving. The header provides the necessary information to locate the first layer of the file that contains the metadata needed for the second layer which contains the actual data of the virtual disk

The tool developed for this research aims to aid the investigator in carving VMDK files, the tool doesn't output a fully carved file, but with the information provided a manual carving can be done, additionally the output of the tool provides additional insight into the structure of the file, it can be applied to both allocated files or to find data in unallocated space. The main identified limitation is that although it can identify the different sparse files, it cannot order them, and this may become a major challenge when multiple VMs exist in a source.

The tool facilitates the handling of the different offsets needed to analyse data within the VMDK files. In a normal operation, the investigator would need to be constantly converting and adding the offsets to move within the file, the tool gives sector and byte offsets for each GD, GT and grain found. The tool can also help to map a known location within the virtual disk to a location inside the sparse VMDK files.

The first contribution of this research is that it expands the current knowledge about the files that comprise a virtual disk and proofs that their structure can be leveraged to carve them. The virtual machine hard disk (VMDK) files present a challenge in which they do not have a flat structure that can be identified by a header and a footer, it is comprised by two dependent layers in its structure. The carving of VMDK files needs to be a two-layer carving; first it needs to recover the metadata part of the file, the GDs and GTs, with this information it then can recover the second layer, the grains; which are the data part of the file and can only be located with the help of the metadata information.

The second contribution is a tool that aids the study of the VMDK files. Its application can be foreseen in two scenarios, it can be applied to existing VMDK files to help with its analysis, or it can be used to look for traces of VMDK files in unallocated space and, if found, use the output to do a manual carving operation.

Future work concerns a deeper development of the tool, where it can output fully carved files based on the information previously obtained. Further exploration is needed to overcome the limitation of how to order the identified sparse files. Moreover, a tool which leverages both the file system metadata and the file structure could represent a bigger improvement for current investigations methods.

Additionally, research on how virtual disks from bare-metal virtualization are implemented using the VMDK file format as starting point is needed to complement the current knowledge about virtualization platforms.

References

[1] M. G. Noblett, “Computer Analysis and Response Team (CART): The Microcomputer as Evidence,” *Crime Lab. Dig.*, vol. 19, no. 1, p. 6, 1992.

[2] DFRWS, “A Road Map for Digital Forensic Research,” *Proc. 2001 Digit. Forensics Res. Work. (*, pp. 1–42, 2001.

[3] Oracle, “1.1. Introduction to Virtualization,” *Oracle VM User’s Guide for Release 3.1.1*, 2013. [Online]. Available: https://docs.oracle.com/cd/E27300_01/E27309/html/vmusg-virtualization.html.

[4] VMware, “What Is a Virtual Machine?,” *ESXi and vCenter Server 5 Documentation*, 2016. [Online]. Available: https://pubs.vmware.com/vsphere-50/topic/com.vmware.vsphere.vm_admin.doc_50/GUID-CEFF6D89-8C19-4143-8C26-4B6D6734D2CB.html.

[5] VMware, “Hypervisor,” *VMware Glossary*. [Online]. Available: https://www.vmware.com/topics/glossary/content/hypervisor.

[6] VMware, “Virtualization Overview- white Paper.” pp. 1–11, 2006.

[7] S. L. Garfinkel, “Carving contiguous and fragmented files with fast object validation,” *Digit. Investig.*, vol. 4, no. SUPPL., pp. 2–12, 2007.

[8] L. Aronson and J. Van Den Bos, “Towards an engineering approach to file carver construction,” *Proc. - Int. Comput. Softw. Appl. Conf.*, pp. 368–373, 2011.

[9] E. Alshammary and A. Hadi, “Reviewing and evaluating existing file carving techniques for JPEG files,” *Proc. - 2016 Cybersecurity Cyberforensics Conf. CCC 2016*, pp. 55–59, 2016.

[10] A. Pal and N. Memon, “The Evolution of File Carving,” no. March, pp. 59–71, 2009.

[11] H. Riaz and M. A. Tahir, “Analysis of VMware virtual machine in forensics and anti-forensics paradigm,” *6th Int. Symp. Digit. Forensic Secur. ISDFS 2018 - Proceeding*, vol. 2018-Janua, no. Isdfs, pp. 1–6, 2018.

[12] M. Hirwani, Y. Pan, B. Stackpole, and D. Johnson, “Forensic Acquisition and Analysis of VMware Virtual Hard Disks,” no. July, pp. 255–259, 2012.

[13] N. J. Healey, O. Angelopoulou, and D. Evans, “A discussion on the recovery of data from a virtual machine,” *Proc. - 4th Int. Conf. Emerg. Intell. Data Web Technol. EIDWT 2013*, pp. 603–606, 2013.

[14] Y. Prayudi, “Virtual Machine Forensic Analysis And Recovery Method For Recovery And Analysis Digital Evidence,” 2018.

[15] S. Lim, B. Yoo, J. Park, K. Byun, and S. Lee, “A research on the investigation method of digital forensics for a VMware Workstation’s virtual machine,” *Math. Comput. Model.*, 2012.

[16] DMTF, “Open Virtualization Format White Paper,” *DMTF Virtualization Manag. Initiat.*, pp. 1–39, 2009.

[17] VMware, “Virtual Disk Format 5 . 0-VMware Technical Note,” pp. 1–14, 2011.

[18] NIST, “Forensic Images for File Carving.” [Online]. Available: https://www.cfreds.nist.gov/FileCarving/index.html.

[19] NIST, “Forensic Images Used for NIST/CFTT File Carving Test Reports.” [Online]. Available: https://www.cfreds.nist.gov/filecarvingtestreports.html.

[20] Office of Law Enforcement Standards of the National Institute of Standards and Technology, “Test Results for Graphic File Carving Tool: PhotoRec v7.0- WIP,” 2014.

[21] CGSEcurity, “PhotoRec - Known file formats,” 2019. [Online]. Available: https://www.cgsecurity.org/wiki/PhotoRec#Known_file_formats.

[22] Nurhayati and N. Fikri, “The analysis of file carving process using PhotoRec and Foremost,” *Proc. 2017 4th Int. Conf. Comput. Appl. Inf. Process. Technol. CAIPT 2017*, vol. 2018-Janua, pp. 1–6, 2018.

Appendix 1 – VMDK carving tool source code

```
"""Imports"""
import argparse
import csv
import hashlib
import io
import struct
import time
from pathlib import Path

SECTOR_SIZE = 512
BLOCK_SIZE = io.DEFAULT_BUFFER_SIZE
HEADER_BIN_SIZE = 128
GRAIN_TABLE_SIZE = 4 * SECTOR_SIZE
GRAIN_SECTORS = 128
GRAIN_COVERAGE = GRAIN_SECTORS * SECTOR_SIZE


def arguments():
    """Return the definition of the arguments for the argument parser"""
    parser = argparse.ArgumentParser(description='VDMK file carving helper')
    parser.add_argument('file', type=str, help='Name of the disk image file to process')
    options_group = parser.add_mutually_exclusive_group()
    options_group.add_argument('-H', '--header', help='Look for possible headers of VMDK files'
                               , action="store_true")
    options_group.add_argument('-D', '--directory', type=str, help='Look for possible grain '
                                                                   'directory data from an input '
                                                                   'file containing headers'
                                                                   ' information',
                               metavar='input_file')
    options_group.add_argument('-T', '--table', type=str, help='Look for possible grain table'
                               , metavar='input_file')
    parser.add_argument('-o', '--output', type=str, required=True,
                        help='Base name for the output file', metavar='basename')
    args = parser.parse_args()
    return args


def arguments_validation(args):
    """"Verifies that arguments have been entered in the command line"""
    if args.header is False and args.directory is None and args.table is None:
        print('Please specify an option to process the file or use -h to see the help\n')


def file_initialization(args_file):
    """Initializes file values"""

    file_size = Path(args_file).stat().st_size
    file = open(args_file, 'rb')
    file.seek(0)
    return file, file_size


def search_headers(image_file, file_size):
    """Search for KDMV string in the image file, outputs offsets of possible headers"""
    read_block = image_file.read(BLOCK_SIZE)
    file_hash = hashlib.sha1()
    blocks = file_size // BLOCK_SIZE
    offset = 0
    position = 0
    possible_headers_offsets = []
    if blocks % BLOCK_SIZE != 0:
        blocks += blocks + +1
    print('Searching...\n')
```

```
    for block in range(blocks):
        file_hash.update(read_block)
        find_offset = -1
        while find_offset == -1:
            find_offset = read_block.find(b'\x4b\x44\x4d\x56', offset, len(read_block))
            if find_offset != -1:
                possible_headers_offsets.append(position + find_offset)
                offset = find_offset + 4
                find_offset = -1
            else:
                read_block = image_file.read(BLOCK_SIZE)
                position += BLOCK_SIZE
                offset = position % BLOCK_SIZE
                find_offset = 0
    return possible_headers_offsets, file_hash.hexdigest()


def create_descriptor_file(header_descriptor_offset, header_descriptor_size, image_file, offset):
    """Create descriptor file if found embedded"""
    if header_descriptor_offset != 0:
        descriptor = image_file.read(header_descriptor_size*512).strip(b'\x00')
        try:
            descriptor_file = open('descriptor_' + str(offset) + '.txt', 'wb')
        except OSError as file_error:
            print(file_error)
        else:
            descriptor_file.write(descriptor)
            descriptor_file.close()
    else:
        pass


def possible_headers_parsing(possible_headers_offsets, image_file, output_file):
    """Parses the possible headers, and outputs the headers which met the conditions
    to be considered valid headers"""
    headers = []
    try:
        headers_output_file = open(output_file + '_headers.txt', 'w')
        headers_bin_file = open(output_file + '_headers.bin', 'wb')
    except OSError as error:
        print(error)
    else:
        for offset in possible_headers_offsets:
            if offset % SECTOR_SIZE != 0:
                pass
            else:
                image_file.seek(offset)
                header = image_file.read(79)
                header_struct = struct.unpack('<3I4QI3Q?4cH', header)
                if (0 < header_struct[1] < 3) and (header_struct[4] == GRAIN_SECTORS) \
                        and (header_struct[7] == 512) and (header_struct[3] % GRAIN_SECTORS == 0):
                    headers.append(header_struct)
                    headers_output_file.write('Header is at byte offset:\t' + str(offset)
                                              + '\n' + 'Header is at sector offset:\t' +
                                              (str(offset//512)) +
                                              '\n' + header_output_formatting(header_struct))
                    header_offset = header + int(offset).to_bytes(8, byteorder='little')
                    headers_output_file.flush()
                    headers_bin_file.write(header_offset +
                                           struct.pack('x' *
                                                       (HEADER_BIN_SIZE - len(header_offset))))
                    headers_bin_file.flush()
                    if header_struct[5] != 0:
                        create_descriptor_file(header_struct[5], header_struct[6], image_file
                                               , offset)
                else:
                    pass
        headers_output_file.close()
        headers_bin_file.close()
        headers_report = '\nFind {0} headers'.format(len(headers))
        if headers:
            headers_report += '\nOutput files:\n\t{0} with hash SHA-1 {1}\n\t{2} with ' \
                              'hash ' \
                              'SHA-1 {3}'.format(headers_output_file.name,
                                                hash_files(output_file + '_headers.txt')
                                                , headers_bin_file.name,
                                                hash_files(output_file + '_headers.bin'))
```

```
        else:
            Path(output_file+'_headers.txt').unlink()
            Path(output_file + '_headers.bin').unlink()

    return headers_report


def header_output_formatting(header_struct):
    """Format the headers for presenting in the output file"""
    header_output = 'Magic number:\t{0}\nVersion:\t{1}\nCapacity:\t{2}\nGrain size:\t{3}' \
                    '\nNumber of GTEs per GT:\t{4}\nRedundant grain directory offset:\t{5}' \
                    '\nGrain directory offset:\t{6}\nOverhead:\t{7}\n\n'\
                    .format(header_struct[0], header_struct[1], header_struct[3], header_struct[4],
                            header_struct[7], header_struct[8], header_struct[9], header_struct[10])
    return header_output


def hash_files(file_name):
    """File hashing function receives file as an input returns the hash"""
    try:
        file = open(file_name, 'rb')
    except OSError as file_error:
        print(file_error)
    else:
        file_hash = hashlib.sha1()
        file_size = Path(file_name).stat().st_size
        blocks = file_size // BLOCK_SIZE
        if blocks < BLOCK_SIZE:
            blocks = 1
        elif blocks % BLOCK_SIZE != 0:
            blocks = blocks + 1
        file.seek(0)
        for block in range(blocks):
            search = file.read(BLOCK_SIZE)
            file_hash.update(search)

    return file_hash.hexdigest()


def headers_file_read(headers_input_file):
    """"Read binary input file and returns all the headers information"""
    try:
        headers_file = open(headers_input_file, 'rb')
        headers = []
    except OSError as error:
        print(error)
    else:
        for index, header in enumerate(struct.iter_unpack('<3I4QI3Q?4cHQ41x',
                                                          headers_file.read(
                                                              Path(headers_input_file)
                                                              .stat().st_size))):
            headers.append(header)
        headers_file.close()

    return headers


def gd_files_generation(header_input_file, output_name, image_file):
    """"Generate the files with the grain directory information"""
    try:
        headers_bin_file = open(header_input_file, 'rb')

    except OSError as error:
        print(error)
    else:
        headers_bin_file.seek(0)
        headers = headers_file_read(header_input_file)
        file_hash = []
        gd_report = '\nOutput files:'
        for header in headers:
            gd_total_entries = ((header[3] // GRAIN_SECTORS) // 512)
            print('\nHeader at:\t{0} with redundant grain directory at\t{1} and' \
                            ' grain directory at\t{2}'.format(header[17],
                                                              header[17] + header[8] * SECTOR_SIZE,
                                                              header[17] + header[9] *
                                                              SECTOR_SIZE))
```

```
            file_hash.append(gd_parsing(image_file, header[17], header[8], header[3],
                                        gd_total_entries, output_name))
            file_hash.append(gd_parsing(image_file, header[17], header[9], header[3],
                                        gd_total_entries, output_name))
        for file in file_hash:
            gd_report += '\n{0}\twith hash SHA-1:\t{1}'.format(file[0], file[1])
        headers_bin_file.close()
    return gd_report


def gd_parsing(image_file, header, gd_offset, disk_size, gd_total_entries, output_name):
    """Parses the grain directory"""
    try:
        gd_csv_file = open(output_name + '_GD_' + str(header * SECTOR_SIZE) + '_' +
                           str(gd_offset * SECTOR_SIZE) + '.csv', 'w', newline='')
        file_hash = [gd_csv_file.name, None]
    except OSError as error:
        print(error)
    else:
        csv_file_writer = csv.writer(gd_csv_file, delimiter=',')
        csv_file_writer.writerow(['Header at byte offset', 'GD at byte offset', 'GD at sector '
                                  'offset', 'GDE', 'GT at byte offset', 'GT at sector offset'])
        image_file.seek(header + (gd_offset * SECTOR_SIZE))
        for index, gde in enumerate(struct.iter_unpack('<I',
                                                       image_file.read(gd_total_entries * 4))):
            if 0 < gde[0] < disk_size:
                csv_file_writer.writerow([header, header + (gd_offset * SECTOR_SIZE),
                                          (header + (gd_offset * SECTOR_SIZE)) // SECTOR_SIZE,
                                          index, header + (gde[0] * SECTOR_SIZE),
                                          (header + (gde[0] * SECTOR_SIZE)) // SECTOR_SIZE])
            else:
                csv_file_writer.writerow([header, header + (gd_offset * SECTOR_SIZE),
                                          (header + (gd_offset * SECTOR_SIZE)) // SECTOR_SIZE,
                                          index, 'Unallocated', 'Unallocated'])
        gd_csv_file.close()
        file_hash[1] = hash_files(file_hash[0])
    return file_hash


def gt_files_generation(header_input_file, image_file, output_name):
    """"Generates the files with the grain table information"""
    try:
        headers_bin_file = open(header_input_file, 'rb')

    except OSError as error:
        print(error)
    else:
        headers_bin_file.seek(0)
        headers = headers_file_read(header_input_file)
        headers_gd = [[header[17], header[8] * SECTOR_SIZE, header[9] * SECTOR_SIZE, header[3]]
                      for header in headers]
        file_hash = []
        gt_report = '\nOutput Files:'
        for gd in headers_gd:
            file_hash.append(gt_parsing(image_file, gd[0], gd[1], gd[3], output_name))
            file_hash.append(gt_parsing(image_file, gd[0], gd[2], gd[3], output_name))
        for file in file_hash:
            gt_report += '\n{0}\twith hash SHA-1:\t{1}'.format(file[0], file[1])
            headers_bin_file.close()
    return gt_report


def gt_parsing(image_file, header, gd_offset, disk_size, output_name):
    """"Parses Grain table"""
    try:
        gt_csv_file = open(output_name + '_GT_' + str(header * SECTOR_SIZE) + '_' +
                           str(gd_offset) + '.csv', 'w', newline='')
        file_hash = [gt_csv_file.name, None]
    except OSError as error:
        print(error)
    else:
        csv_file_writer = csv.writer(gt_csv_file, delimiter=',')
        csv_file_writer.writerow(
            ['Header at byte offset', 'GD at byte offset', 'GDE', 'GT at byte offset',
             'GT at sector offset', 'Grain', 'Grain starts at byte offset',
             'Grain starts at sector offset', 'Grain ends at byte offset',
             'Grain ends at sector offset'])
        image_file.seek(header + gd_offset)
```

```
        for gde_index, gde in enumerate(struct.iter_unpack('<I', image_file.read(SECTOR_SIZE))):
            for gte_index, gte in enumerate(struct.iter_unpack('<I', image_file.read(
                    GRAIN_TABLE_SIZE))):
                if 0 < gte[0] < disk_size:

                    csv_file_writer.writerow([header, header + gd_offset, gde_index,
                                              header + (gde[0] * SECTOR_SIZE),
                                              (header + (gde[0] * SECTOR_SIZE)) // 512, gte_index,
                                              header + (gte[0] * SECTOR_SIZE),
                                              (header + (gte[0] * SECTOR_SIZE)) // SECTOR_SIZE,
                                              (header + (gte[0] * SECTOR_SIZE)
                                               + GRAIN_COVERAGE) - 1,
                                              (header + ((gte[0] * SECTOR_SIZE) + GRAIN_COVERAGE)
                                               - 1) // SECTOR_SIZE])
                else:
                    csv_file_writer.writerow([header, header + gd_offset, gde_index,
                                              header + (gde[0] * SECTOR_SIZE),
                                              (header + (gde[0] * SECTOR_SIZE)) // 512, gte_index,
                                              'Unallocated', 'Unallocated', 'Unallocated',
                                              'Unallocated'])
        gt_csv_file.close()
        file_hash[1] = hash_files(Path(file_hash[0]).name)
    return file_hash


def get_case_info():
    """Get case information from user input"""
    case_info = [['Case Number:\t', ''], ['Evidence number:\t', ''],
                 ['Unique description:\t', ''], ['Examiner:\t', ''], ['Notes:\t', '']]
    print('Please provide evidence item information')
    for info in case_info:
        info[1] = input(info[0])
    return case_info


def report(case_info, phase, report_filename):
    """Generate file for final report"""
    try:
        file = open(report_filename, 'w')
    except OSError as error:
        print(error)
    else:
        file.write('VMDK file carving helper - processing report\n\n')
        for info in case_info:
            file.write(info[0] + info[1] + '\n')
        file.write(phase)
    file.close()


def main():
    """Main function"""
    args = arguments()
    arguments_validation(args)
    case_info = get_case_info()
    try:
        start_time = time.time()
        image_file, image_file_size = file_initialization(args.file)
        print('\nImage file: {0}\nFile size: {1:,} bytes'.format(image_file.name, image_file_size))
    except OSError as file_error:
        print(file_error)
    else:
        if args.header:
            possible_headers_offsets, image_hash = search_headers(image_file, image_file_size)
            headers_report = '\nImage file: {0}\nFile size: {1:,} bytes\nHash SHA-1:\t{2}' \
                .format(image_file.name, image_file_size, image_hash)
            headers_report += possible_headers_parsing(possible_headers_offsets, image_file,
                                                       args.output)

            end_time = time.time()
            headers_report += '\n\nProcess started at:\t{0}\nProcess ended at:\t{1}'.format(
                time.ctime(start_time), time.ctime(end_time))
            print(headers_report)
            report(case_info, headers_report, args.output + '_report' + '.txt')
```

```
        if args.directory:
            headers_bin_file_size = Path(args.directory).stat().st_size
            print('\nNumbers of headers to process: {0}'.format(headers_bin_file_size //
                                                               HEADER_BIN_SIZE))
            gd_report = gd_files_generation(args.directory, args.output, image_file)
            end_time = time.time()
            gd_report += '\n\nProcess started at:\t{0}\nProcess ended at:\t{1}'.format(
                time.ctime(start_time), time.ctime(end_time))
            print(gd_report)
            report(case_info, gd_report, args.output + '_GD_report' + '.txt')

        if args.table:
            headers_bin_file_size = Path(args.table).stat().st_size
            print('\nNumbers of headers to process: {0}\nProcessing...'
                  .format(headers_bin_file_size // HEADER_BIN_SIZE))
            gt_report = gt_files_generation(args.table, image_file, args.output)
            end_time = time.time()
            gt_report += '\n\nProcess started at:\t{0}\nProcess ended at:\t{1}'.format(
                time.ctime(start_time), time.ctime(end_time))
            print(gt_report)
            report(case_info, gt_report, args.output + '_GT_report' + '.txt')
        image_file.close()

    finally:
        print('\nProcess finished')


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

Appendix 2 – TC – 01 report

```
VMDK file carving helper - processing report

Case Number:      001
Evidence number:  001
Unique description:     TC-01
Examiner:   Raul Jimenez
Notes:      Test case 01, clean system

Image file: .\TC-01.000
File size: 250,059,350,016 bytes
Hash SHA-1: 0c28eb77cc73dffc06d82353f9e72279111c8726
Find 0 headers

Process started at:     Sun Mar 31 02:52:54 2019
Process ended at: Sun Mar 31 04:28:08 2019
```

Appendix 3 – TC – 02 header summary file

```
Header is at byte offset:     3170103296
Header is at sector offset:   6191608
Magic number:     1447904331
Version:    1
Capacity:   327680
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 22
Overhead:   128

Header is at byte offset:     38250364928
Header is at sector offset:   74707744
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     40761458688
Header is at sector offset:   79612224
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     42125635584
Header is at sector offset:   82276632
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     42841239552
Header is at sector offset:   83674296
Magic number:     1447904331
Version:    1
Capacity:   8323072
```

```
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     44008906752
Header is at sector offset:   85954896
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024
```

Appendix 4 – TC – 03 header summary file

```
Header is at byte offset:     38250364928
Header is at sector offset:   74707744
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     40761458688
Header is at sector offset:   79612224
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     42125635584
Header is at sector offset:   82276632
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     42841239552
Header is at sector offset:   83674296
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     44008906752
Header is at sector offset:   85954896
Magic number:     1447904331
Version:    1
Capacity:   8323072
```

```
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024
```

Appendix 5 – TC – 04 header summary file

```
Header is at byte offset:     1715712000
Header is at sector offset:   3351000
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     1715732480
Header is at sector offset:   3351040
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     1761800192
Header is at sector offset:   3441016
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     1762050048
Header is at sector offset:   3441504
Magic number:     1447904331
Version:    1
Capacity:   8323072
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 510
Overhead:   1024

Header is at byte offset:     1762058240
Header is at sector offset:   3441520
Magic number:     1447904331
Version:    1
Capacity:   327680
```

```
Grain size: 128
Number of GTEs per GT:  512
Redundant grain directory offset:   1
Grain directory offset: 22
Overhead:   128
```