Communications on Applied Electronics (CAE) – ISSN : 2394-4714
Foundation of Computer Science FCS, New York, USA
Volume 7 – No. 9, November 2017 – www.caeaccess.org

Unconventional Method of Accessing Files – An Automated Generation of its Input

K. Srinivas
Research Scholar in CSE
JNTUH University, Kukatpalli,
Hyderabad, Telangana, India

T. Venugopal, PhD
Professor, Department of CSE
JNTUH College of Engineering,
Jagtial Dist. Telangana, India

ABSTRACT
File Carving is an unconventional method of accessing files from disk. It is a technique of reassembling unordered mixed file fragments, without using files' metadata such as FAT, for reconstructing the actual files present on the disk. In the areas of data recovery and digital forensics this situation arises. A challenge file is an input file for testing a file carving tool during its development phase and it consists of a number of files, in the form of fragments, mixed in random order [1]. In this paper authors have presented a software system that generates a challenge file by implementing, at user level, a file system which broadly follows FAT file system. This software system uses a large size file to store file fragments just like a kernel level file system uses disk to store files. The designers of file carvers can use the challenge file conveniently as a virtual disk, in place of the actual disk, thus eliminating the need of a physical hard disk for testing their algorithms. The kernel level file system fragments the file, as per availability of free clusters, at the time of creation or modification of files. The user level file system, fragments the file, as per availability of free clusters, on the virtual disk i.e., the challenge file. This challenge file consists of mixed file fragments of a number of user files. There are a number of other benefits of this approach as outlined in this paper.

General Terms
File Carving, Digital Forensics, Data Recovery, File System, Disk Clusters, Operating Systems.

Keywords
User Level File System, Kernel Level File System, Virtual disk, Challenge file, Script File.

1. INTRODUCTION
When a user saves a file on a disk, the Operating System uses its File System component to handle it. A File System is a set of software modules, at kernel level, for file handling operations. Assume that a user has created a file named as "one.txt" containing the text "abc". The size of this file is 3 bytes. The file system allocates one free clusters for this new file, at the time of its creation, from the pool of free clusters that it maintains. A cluster is a set of consecutive sectors on the disk. A cluster is an allocation unit. The kernel file system views the disk as a set of clusters than as a set of bytes. When a user creates a new file, the required number of free clusters is allocated for it. And when a user deletes a file, all the used clusters by the file are freed. So, for the above "one.txt" file, one cluster is allocated. Thus when the properties of the file "one.txt" are viewed on Windows 7 Operating System, we notice file size as 3 bytes and size on the disk as 4096 bytes. In this paper we assume the cluster size as 4096 because it is the most common size but it can vary [2].

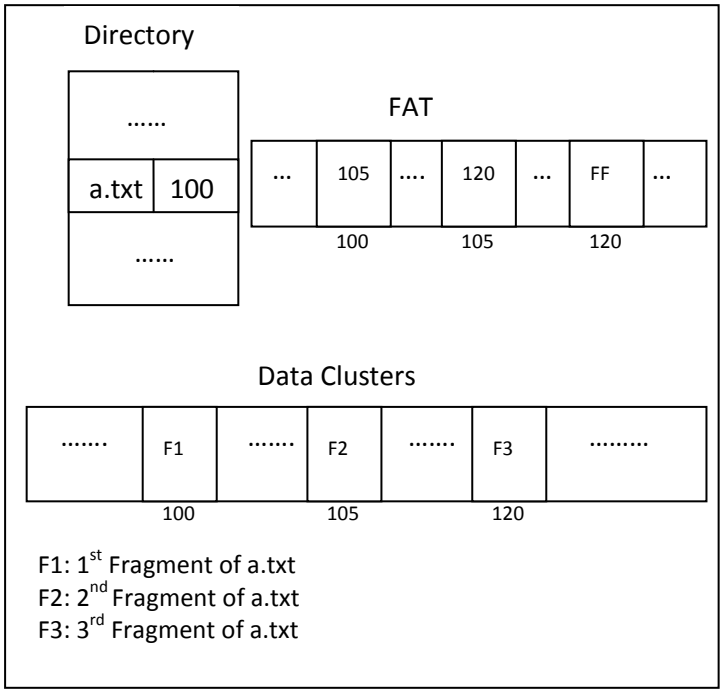

Figure 1. The three regions of a Hard Disk namely directory, FAT and data clusters, used by Operating System in Conventional Method of Accessing a Fragmented File

Consider a file named "a.txt" of size 10KB on the disk saved at cluster numbers 100, 105 and 120. In *conventional method*, to perform read operation on this file, the file system obtains these cluster numbers (that were saved in the file system's data structures when the file was created, as shown in Figure 1), reads data from these clusters and presents the data to user application [4]. It is up to user application how to interpret this data.

When the file is deleted, the File System changes the first byte of the file name of a.txt to '_'. Then it stores a zero in each of the FAT locations at indexes 100, 105 and 120 to indicate that the clusters 100, 105 and 120 are free now. The actual data of a file a.txt is *not erased* [4]. In an unconventional method of accessing files, file fragments are to be reassembled in the absence of metadata in file system data structures as shown in Figure 2. Unconventional methods are applied under three different situations. A) When files were deleted accidentally and they need to be recovered. B) When file(s) were deleted

by a criminal intentionally to escape from the law for his criminal activities and investigating agencies want to view such files [3]. C) When file system data structures such as DIR and FAT got corrupted and the files present on the disk to be read. Under these three conditions the conventional method cannot be used.

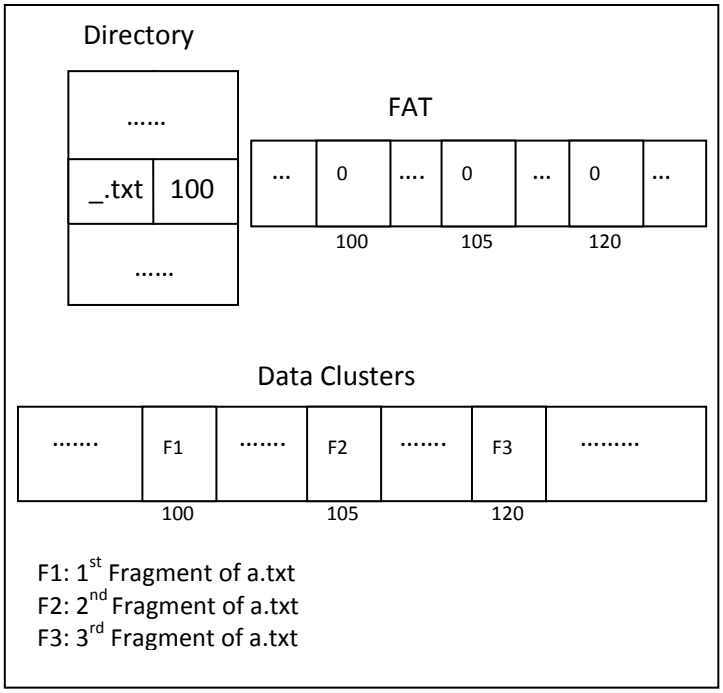

Figure 2. The three regions of a Hard Disk namely directory, FAT and data clusters, after deleting the a.txt File. Operating System cannot access the file now. Unconventional Method of Accessing the Fragmented File a.txt is needed, if it is forensically important

To face the above situations technically in the areas of data recovery and digital forensics, a new technology known as file carving has evolved. File carving is a technique of identifying, reassembling mixed file fragments, in the absence of files' metadata, to reconstruct the actual files present on the disk [4].

In a conventional method of reading a file, the file system refers to file-system's data structures, reads the data present in clusters and then presents it to the application at user level. The conventional methods are not useful for accessing the files present on the disk in situations described in the above paragraphs. File carving is an unconventional method of accessing files from the disk when the files' metadata is not available.

A file carving algorithm, during its development phase, needs to test a used disk partition of small size to verify the correctness of the developed algorithm. As a replacement for the test disk, in recent research, a large file containing mixed file fragments but not containing the metadata is used to verify the correctness of the developed file carving algorithms [1].

When a used disk partition is considered, it contains naturally file fragments because the file system always cannot allocate clusters in contiguous area on the disk for a file [5]. The same state is created *artificially* on a large file and is used as an alternative to a test disk [1, 7]. This large file therefore contains unordered mixed file fragments of a number of files without any metadata of files. It is a challenge for a file carving algorithm to join these pieces for reconstructing the actual files present in it and present to the user applications. Therefore, a large file containing unordered, mixed file fragments of a number of files without any files' metadata is called a challenge file. A tool for an automated generation of a challenge file that is as natural as a used disk partition is required to provide realistic data sets that are as complex as the data on real used disks [6]. In this paper, authors present a software system that implements an automated construction of a challenge file that is as natural as a used disk partition.

The presentation of our work is planned, in this paper, as follows. In section II, a structure of a challenge file is described. In section III, the principles adopted for an automated creation of a challenge file are described. In section IV, design and implementation of software system is presented. In section V, results of our experiments are presented. Finally in section VI, the number of benefits of this approach is described.

2. STRUCTURE OF A CHALLENGE FILE

The automated construction of a natural challenge file consists of the THREE phases. 1) The initial phases 2) Construction phase 3) Fine-tuning phase. In this section, the structure of a challenge file during each of the above phases is described.

2.1 Structure of a Challenge File in Initial Phase

In an initial phase, the challenge file is viewed as consisting of contiguous-clusters with each cluster containing *all zeros* in its System Area i.e. directory and FAT regions. A cluster is a set of 4096 contiguous bytes *starting* at a byte offset satisfying the equation (1). The above state of a challenge file is equivalent to a disk containing *no files* on it and files system data structures in resonance with the same. The structure of a challenge file thus is as shown in Figure 3.

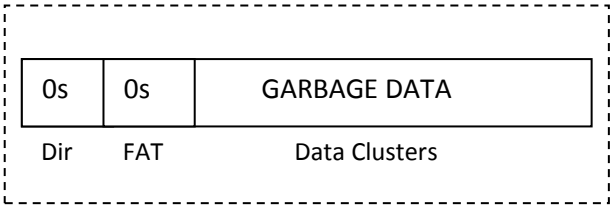

Figure 3. The Structure of a Challenge File in an Initial Phase

We note that every file fragment of a user file in a Challenge-file should start at a byte offset satisfying Equation (1). The reason is that, on a disk, every fragment of a file starts at a byte offset satisfying the Equation (1).

$$((\text{OFFSET} \ \% \ \text{CS}) = 0) \qquad (1)$$

On a disk, let the space occupied by a file of size (F_S) be FD_S which is always equals to multiple of cluster size (CS) as given in Equation (2) even though the actual size is not multiple of cluster size (CS). In Equation (2), % represents mod operation, division operation returns quotient i.e. an integer and ? represents 'if' condition as in C source language.

$$FD_S = \frac{F_S}{\text{CS}} + (F_S \,\%\, \text{CS})\ ?\ 1 : 0 \qquad (2)$$

2.2 Structure of Challenge File in Construction Phase

During the construction phase, the challenge file consists of three regions as shown in Figure 4. The three regions are; *A)* DIR *B)* FAT *C)* Data-Clusters as described in section I. The state of a challenge file is equivalent to a disk containing many files of which three files File-A, File-B and File-C such that the File-A and File-B (fragmented files) and File-C (the contiguous file) requiring 4 clusters, 3 clusters and 2 clusters respectively.

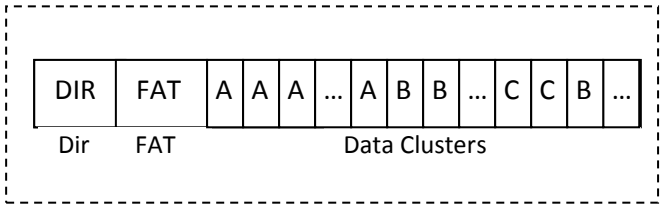

Figure 4. The Structure of a Challenge File in Construction Phase

The three files together require 9 clusters. Therefore nine corresponding locations of FAT store non-zero values. All FAT locations corresponding to free clusters store zeroes. The three entries in DIR are allocated one for each of the three files File-A, File-B and File-C. The total number of allocated entries in the DIR region is equal to the number of files created on the disk.

2.3 Structure of a Challenge File after Fine-Tuning Phase

The structure of a challenge file after the fine-tuning phase is shown in Figure 5. It is equivalent to a 'used disk' containing data of files created during the construction phase but not containing metadata of all the files created during the construction phase. Therefore, in Figure 5 data of File-A, File-B and File-C is present but does not contain their metadata.

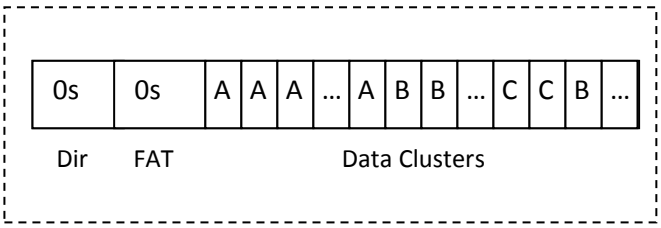

Figure 5. The structure of a Challenge File after Fine Tuning Phase

The above challenge file is a challenge for a file carving algorithm., During its development phase, coding of File carving algorithm can be verified about its correctness in joining pieces for retrieving File-A, File-B, File-C and others. Submitting a challenge-file to a file-carver developer is equivalent to submitting a disk to a digital forensic expert or a data recovery expert. The challenge file acts as a used disk for file carving algorithms and therefore we use virtual disk as a synonym for the term challenge file.

3. THE TECHNIQUE OF CONSTRUCTION OF A CHALLENGE FILE

The technique of construction of a challenge file (i.e., virtual disk) in each of the three phases is presented below.

3.1 The Technique of Construction in Initial Phase

A virtual disk (i.e. a challenge-file) of user-specified size S GB is created in binary mode and initialized to contain all zeroes in System Area (DIR + FAT regions). Depending upon the size of the virtual disk, the size of the DIR region and the size of the FAT region are calculated. The various parameters for the virtual disk are specified in the Table 1.

The cluster size is decided as 4KB as this is the common size used in Operating Systems. Each directory entry contains two fields namely filename and starting cluster. The directory entry size is 16 bytes because the filename maximum size is decided as 14 bytes and 2 bytes for starting cluster number. A 2-byte unsigned number is used to represent a cluster number and hence the maximum number of clusters supported is 2^{16} clusters.

Let CS, CL_{MAX} and VDS_{MAX} be the cluster size, maximum number of clusters and maximum size of the virtual disk respectively. Then maximum size of the virtual disk that system can support is given by Equation (3). The maximum size of the virtual disk is 0.25 GB when cluster size is 4KB and each cluster is represented by 16-bits.

$$VDS_{MAX} = (CL_{MAX} * \ \text{CS}) \qquad (3)$$

Equation (4) gives the maximum number of directory entries (DE_{MAX}), as a function of Cluster Size (CS) and each Directory Entry Size (DES). The maximum number of directory entries per cluster DE_{MAX} is 256 when cluster size is 4KB and each directory size is 16 bytes.

$$DE_{MAX} = \frac{CS}{DES} \qquad (4)$$

The specification of a virtual disk with various parameters is presented in Table 1.

TABLE 1. The Specifications of Virtual Disk

Cluster size	4KB
Each FAT entry size	2 bytes
Maximum clusters	2^{16} clusters
Max Virtual Disk Size	0.25 GB
Each DIR entry size	16 bytes (14+2)
Max no of DIR entries/ cluster	256 entries
Max no.of FAT entries/cluster	2K entries
minimum file size on disk	4KB

3.2 The Technique of Construction in Construction Phase

In this phase, the user performs a sequence of command operations of his choice. Each command is selected from the following list; A*)* Create a new file *B)* Modify an existing file *C)* Delete a file. The result of the sequence of operations is that the user files are saved on the virtual disk (i.e., the challenge file) in fragmented/contiguous areas depending upon the availability of free clusters for each file just like the kernel level file system saves files on a *real* disk. Therefore

natural fragmentation is achieved. A script file containing the sequence of commands automates the above operation. Therefore automated generation of a natural challenge file is achieved. The challenge file contains metadata during this phase.

3.3 The Technique of Construction in Fine-Tuning Phase

In this phase, the DIR and the FAT regions of the virtual disk are erased so that the virtual disk is a real challenge for the file carving algorithm.

After the three phases of construction are over the result is a challenge file. It contains data of user files in data clusters but does not contain any metadata. The script file automates generation of this challenge-file. The files are naturally fragmented just like a file-system fragments files on the disk because the *similar* file-system is used but in user space.

4. DESIGN AND IMPLEMENTATION

This software system consists of the following four classes; *1)* DIR class *2)* FAT class *3)* FileSystem Class *4)* UserSpace Class. In this section, the responsibilities assigned to these classes are described in detail.

4.1 The DIR Class

The DIR class is assigned with the following responsibilities. 1) Write a Directory entry 2) Obtain a free directory entry 3) Obtain a start cluster of a given file 4)Make free an existing directory entry.

Each directory entry consists of two fields. They are filename and starting cluster number. The maximum size of a filename is fixed as 14 characters. A two-byte integer number is used to represent one cluster number. So each directory entry is 16 bytes. The member function is supposed to prepare the record containing these two fields and should write this record to the first free entry in the directory table. A slot in a directory table is said to be free if it's starting cluster number is zero or filename starting byte is '_'. This operation needs to be performed when a new file is created on a virtual disk.

To read a file present on the virtual disk, in conventional method, we need to know the fat chain of a file. The fat chain starting cluster is present in the directory region and the actual chain is present in the fat region. The end of the fat chain is marked as a number 0xFFFF. The initial cluster number is obtained from the file's corresponding entry in the directory region.

When a file is deleted, the corresponding entry in the directory region must be made free. When the first byte of a filename, in the directory entry, is changed to '_', it marks a free cluster. When the directory region is initialized the whole of it is written with all zeroes. It is not required to make the starting cluster as zero for an entry corresponding to the file being deleted.

In a kernel level file-system, each entry in a directory table contains other attributes like file size, time of file creation etc. From file-carving point of view, it is required to introduce fragmentation naturally and therefore it does not require storing all the attributes of a file in directory entry. So the fields in a directory are restricted to filename and starting-cluster number in user space file-system.aptions should be Times New Roman 9-point bold. They should be numbered (e.g., "Table 1" or "Figure 2"), please note that the word for Table and Figure are spelled out. Figure's captions should be centered beneath the image or picture, and Table captions should be centered above the table body.

4.2 The FAT Class

The FAT class is assigned with the following responsibilities. 1) Get the required number of free clusters 2) Given a set of cluster numbers, form a fat chain 3)Obtain the fat chain given a starting cluster number.

The FAT region is basically an array, each element occupying 2 bytes. When the FFAT entry is zero, then the corresponding cluster is a free cluster. When a new file is created or when an existing file is extended, we need a certain number of free clusters on the virtual disk so that the new file data or extended file data is written to these clusters. After writing data to the free clusters, a fat chain needs to be formed. If a file's data is written to cluster numbers 10, 105 and 120 then at the 10^{th} location the element 100 is written. At 100^{th} location the element 105 is written. At 105^{th} location, the element 120 is written. At 120^{th} location, the element 0xFFFF is written. The value 0xFFFF marks the last cluster in the chain. Suppose that a file is stored at cluster numbers 10, 105 and 120. When a file is to be read, starting cluster number is obtained from a directory region. The value 10 is obtained from the corresponding entry in a directory region. Then in the fat region, the element at 10^{th} location is read. Its value is 105. Then the element at 105^{th} location is read. Its value is 120. Then the element at location 120 is read. The value at this location is 0xFFFF. So the fat chain is 10->105->120. So it reads data from cluster numbers 10, 105 and 120.

4.3 The FileSystem Class

The FileSystem class is assigned with the following responsibilities. 1) Read the cluster data given a cluster number. 2) write data to the specified cluster.

Irrespective of file type, the file system always views the file content as a sequence of bytes. To read a file, a starting cluster is obtained from a directory entry. Then the fat chain is obtained from the fat region. Then one-by-one cluster data is read to present it to the user application. So it is required to have a facility in User Space File System (USFS) class that reads the cluster data given a cluster number. When a new file is created, it obtains a free directory entry, finds the number of required clusters, forms fat chain and then writes data to the data clusters. Therefore, in File System class it is required to have one facility for writing the file's bytes to the specified cluster.

4.4 The UserSpace Class

The UserSpace class is assigned mainly with the following responsibilities; *1)* Create a file *2)* Modify a file *3)* Delete a file. These operations are implemented using the facilities provided by the above three classes.

To *create a file* F of size S requiring N number of clusters *on virtual disk* the following steps are executed.

1. N = S / CLSIZE + (S%CLSIZE)?1:0
2. Using *get_free_cls()* of FAT class, obtain N number of free clusters C_0, C_1, C_2, … , C_{N-1}.
3. Read file F's data 4096 bytes at a time using *the kernel level file-system* and write to C_i (i=0,1,2, … , N-1) of virtual disk using *write_cl()* of FileSystem class.

To *delete a file* F of size S requiring N number of clusters *on virtual disk* the following steps are executed.

1. Locate the directory entry of F using *get_dir_entry()*.
2. Modify the first byte of the file name in the directory entry to '_'
3. Make, all the clusters used by the file F, free using *free_next_cls()* of FAT class.

To modify a file F_{before} (of size S_{before} requiring N_{before} clusters) to F_{after} (of size S_{after} requiring N_{after}) on a virtual disk, there are two cases namely *1)* Extending the file *2)* Shrinking the file

Case 1: Extending the file F ($N_{before} < N_{after}$)

1. Locate the directory entry of F_{before} using *get_dir_entry()*.
2. Find the fat-chain of F_{before} using the *get_next_cls()* of the FAT class.
3. Calculate additional number of free clusters required as (N_{after}-N_{before}) and procure the free clusters by using the *get_free_cls()* of the FAT class.
4. Prepare a FAT chain containing the N_{after} number of clusters using *set_next_cls()* of FAT class.
5. Read the file F_{after}, 4KB of data at a time using the *kernel level file system* and write to C_i (i=0,1,2, … ,N_{after}-1) of virtual disk using *write_cl()* of FileSystem class.

Case 2: Shrinking the file ($N_{before} > N_{after}$)

1. Locate the directory entry of F_{before} using *get_dir_entry()*.
2. Find the fat chain of F_{before} using *get_next_cls()* of FAT class.
3. Find additional number of clusters as (N_{before}-N_{after}) and make them free by retaining the initial clusters of previous fat chain.
4. Prepare a FAT chain containing N_{after} number of clusters using *set_next_cls()* of FAT class.
5. Read file F_{after} data 4KB at a time using *kernel level file system* and write to C_i (i=0,1,2, … ,N_{after}-1) of virtual disk using *write_cl()* of FileSystem class.

The programming language C++ is used to implement the above classes. The four C++ classes are given below.

```
class DIR
{
        public:
        int write_dir_entry(char *fn , unsigned int sc) ;
        unsigned int get_free_entry() ;
        unsigned int get_start_cluster(char *fn) ;
        int make_dir_entry_free(char *fn) ;
} ;

class FAT
{
public:
        ui get_free_cl() ;
        ui get_next_cl(ui clno) ;
        ui get_next_cls(ui clno , ui cls[] ) ;
        void set_next_cl(ui clno , ui nextcl) ;
        ui FAT::operator [] (ui clno);
        int get_free_cls(u icls[] , ui n) ;
        void set_next_cls(uicls[] , ui n) ;
        void free_next_cls(ui cls[] , ui clcnt) ;
} ;

class FileSystem
{
public:
        void read_cl(ui clno , uc b[CLSIZE] , ui bytecnt) ;
        void write_cl(ui clno , uc b[CLSIZE] , ui bytecnt) ;
} ;
class UserSpace
{
        DIR dir ;
        FAT fat ;
        FileSystemfs ;
        char command[60] ;
        charcmd[20], char arg1[20], char arg2[20] ;
public:
        voidshowfile() ;
        void format() ;
        voidhdump(ullow=0 , ul high=240) ;
        void write(char fn[]) ;
        void directory() ;
        ui del() ;
        void modify() ;
        UserSpace() ;
        void create() ;
        void help() ;
} ;
```

Implementation of the two important operations of FileSystem class is explained now. The read_cl()member function reads a specified cluster from a challenge file into a buffer b. Using fseek() the file pointer is made to point to the starting byte of the specified cluster. fread() is used toread the cluster of size 4KB into buffer b. The write_cl() member function writes a buffer 'bytes' to a specified cluster on virtual disk. Using fseek() the file pointer is made to point to the starting byte of the specified cluster. fwrite() statement writes the buffer of size 4KB to the virtual disk.In the above two member functions and all other applicable member functions, "test.dat" is a virtual disk and acts as a challenge file after completing the three phases of construction.

```
voidFileSystem :: read_cl(uiclno , uc b[] , uibytecnt)
{
        FILE *fp = fopen("vdisk" , "rb") ;
        fseek(fp , CLSIZE * (long)clno , 0) ;
        fread(b , 1 , bytecnt , fp) ;
        fclose(fp) ;
}
```

The program utilizing the above classes is compiled and executed using Turbo C++ compiler and the results of the experiments are presented in the next section.

5. EXPERIMENTS AND RESULTS

The various operations supported by User Space File System (USFS) software for the user are shown in the screenshot in Figure 6. It also shows the response of "directory" command. In the screenshot, vd> is a prompt for the user. The prompt vd is a short form for virtual disk. "?" is a command to display a list of commands that are available for the user to execute.

Communications on Applied Electronics (CAE) – ISSN : 2394-4714
Foundation of Computer Science FCS, New York, USA
Volume 7 – No. 9, November 2017 – www.caeaccess.org

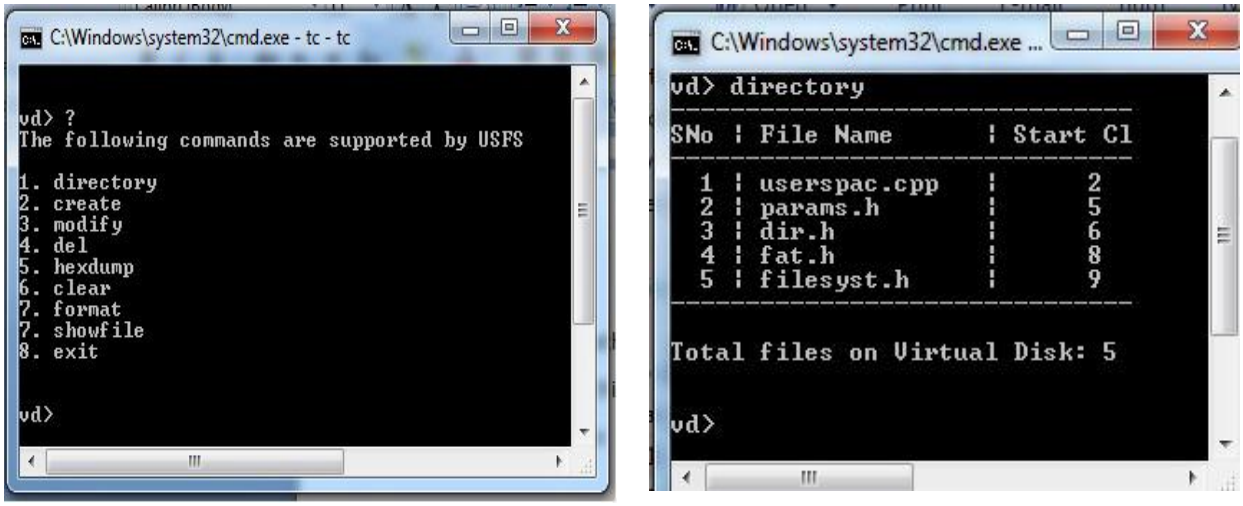

Figure 6. Responses of "?" and "directory" commands

When "del" command is executed, it can be seen that the total files is 5. And it can also be seen that the deleted filename's starting byte is changed to "_". This is the procedure adopted in FAT file system.

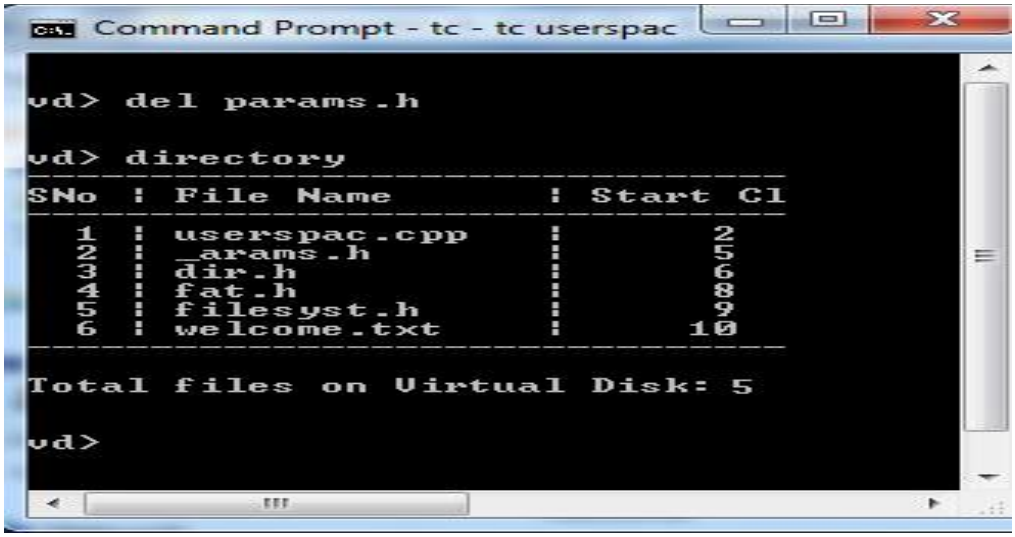

Figure 7. Screen shot of "directory" command after "del" command

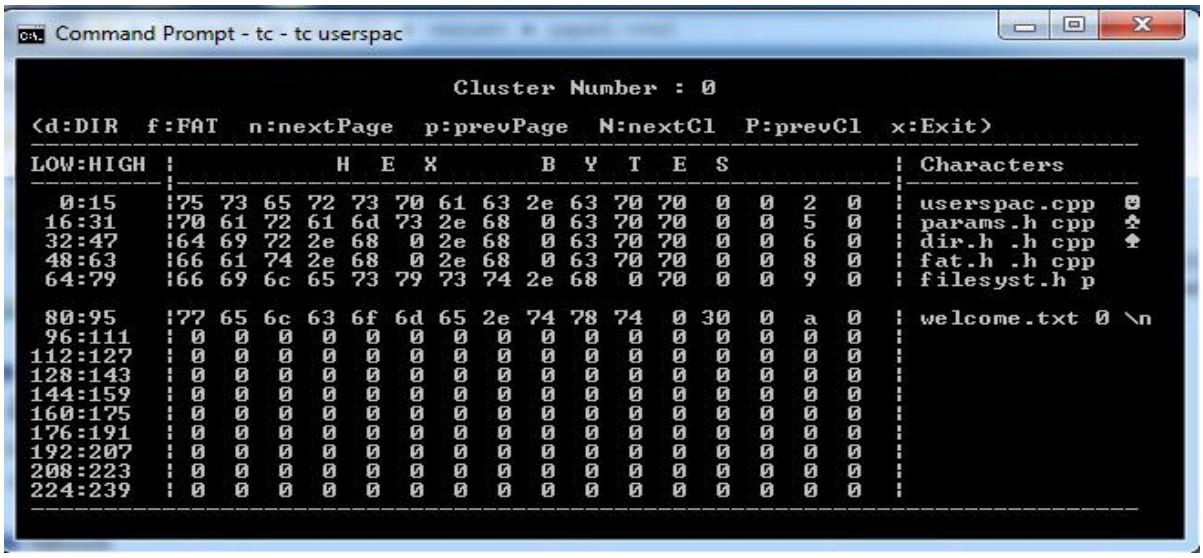

Figure 8. Response of "hexdump" command showing subcommands for browsing through the virtual disk content and showing the selected page in three sections (offset range, hex bytes and character-equivalent of each byte)

In Figure 8, screenshot of "hexdump" command is shown. It contains the cluster number at the top followed by a list of sub commands and followed by a table showing the content of challenge file. The table has 3 columns. The first column gives offset range. The second column displays content in the specified offset range, in hex formats. In third column, same content in the form of text is displayed. The table 2 gives the list of sub commands of "hexdump" command and their actions. These sub-commands help file carver developer to view different portions of the challenge file. For example, user may want to view a specific portion of a cluster because some file carvers compare ending portion of cluster I with starting portion of cluster J [3,8].

Table 2. Subcommands of 'hexdump' command

Command	Action
d	To display **d**irectory region
f	To display **f**at region
n	To display **n**ext page
p	To display **p**revious page
N	To display **N**ext Cluster
P	To display **P**revious Cluster

Finally "format" command is executed as part of the fine-tuning phase that erases the metadata. A disk containing files content but not containing the metadata of files is the prime condition for any file carving algorithm.

6. BENEFITS OF AUTOMATED GENERATION

- Avoid purchase of physical used disks.
- Disk size flexibility
- Natural generation of challenge file
- Avoid artificial /manual generation of challenge file
- Various scenarios can be generated on each virtual disk
- Number of virtual disks can be created
- The virtual disk can be zipped and sent easily over an internet
- Avoids read/write operations on physical disk in raw mode
- Researchers can implement their new ideas of file system without the need to work at kernel level
- Before the final step of automated construction, the directory and fat regions contain the correct answer to the proposed challenge.
- International workshops like DFRWS can create challenge files using USFS and compare participants results with the correct results available in the directory and fat regions before fine-tuning step
- In-place file carving is a technique that avoids creation of carved files on host disk [9]. Such techniques can be tested conveniently using this tool by writing appropriate data directory and fat regions of virtual disk.
- The source code can be used on any platform where as FUSE and Kernel level programming are specific to Linux

7. CONCLUSION

Every file carver algorithm attempts to carve a file of specific type from file fragments without using file's metadata from file system data structures. For conducting experiments to ascertain the correctness of the new algorithms designed by researchers, they need a test hard disk, in which, the test patterns are saved and used as inputs for the program that implements the algorithm developed. Every file carver tool treats the disk under examination in raw mode. And hence computer system's hard disk cannot be used directly for safety of the data present on it during the testing experiments. The User Space File System (USFS) of this paper avoids the purchase of the test disk as well as avoids the accessing of the hard disk of the computer system in raw mode, by creating a virtual disk on computer system's hard disk and fully utilizing the kernel file systems APIs for accessing the virtual disk. This is safe and cost effective approach for providing the input scenarios during development phase of a file carver tool for professionals and a file carver algorithm for researchers to conduct experiments. A virtual disk can be created containing fragmented or contiguous files of file type of user's choice by executing simple commands; create, delete and modify commands. At the same time, layout of the metadata and user data on the disk (i.e. virtual disk) can be seen by using hexdump command. "hexdump" command provides white box approach for the complete disk data. Hence this can be utilized by professionals and researchers to understand the input data applied to file carver tool. So the correct working of tool or algorithm is possible as the command allows the browsing through the data on every part of the virtual disk.

In future work, authors would like to improve User Space File System (USFS) tool in the following aspects. Firstly, during browsing operation, raw data is provided to the user of the tool and the user has to make out what the data means. For example, using hexdump, one can come to know whether a given file is saved on the virtual disk as fragmented or saved on contiguous region of the virtual disk, by browsing through the various sections of the virtual disk. But a command that tells the user whether a given file is fragmented or not, provides more comfort or beneficial. Secondly, user of the tool may want to know the list of cluster numbers consumed by a given file on the virtual disk. Though this information can be obtained by browsing through various sections of the virtual disk, it will provide more comfort to the user if there is a command that gives the list of cluster numbers that are used for saving the file's data. Thirdly, a provision that allows to execute a batch file or script file containing a predetermined set of commands to create a specific scenario, is more appropriate because this approach allows the user to create multiple virtual disks each having one scenario or number of scenarios as per the requirements of the user, for effective utilization of user's time. Finally, authors would like to illustrate how to utilize USFS tool for automated generation of input for unconventional method of accessing image files with the help of a simple file carving tool presented in [7].

8. ACKNOWLEDGMENTS

We would like to convey our sincere thanks to all the staff members of Sree Chaitanya College of Engineering, Karimnagar, India for extending their full cooperation during the writing of this paper. Also we express our gratitude to all the staff members of CSE at JNTUH College of Engineering, Jagtial for their suggestions and continuous support during the period of writing of this paper. Finally we thank our family members, without whom this work would not have been possible, for their patience and constant support.

9. REFERENCES

[1] https://www.dfrws.org

[2] Andreas Dewald, Sabine Seufert, "AFEIC: Advanced forensic Ext4 inode carving",DFRWS 2017 Europe – Proceedings of the 4th Annual DFRWS Europe – Elsevier Journal - Digital Investigation 20 (2017) S83-S91

[3] Nasir Memon, Anandabrata Pal, "Automated Reassembly of File Fragmented Images Using GreedyAlgorithms",IEEE Transactions onImage Processing, Volume 15,No.2, February,2006

[4] Anadabrata Pal, Nasir Memon, "The Evolution of File Carving: The benefits and problems of forensics recovery",IEEE Signal Processing magazine Vol. 26. No 2. March 2009.

[5] Simson L. Garfinkel: Carving contiguous and fragmented files with fast object validation, ELSEVIER, Digital Investigation, 2007.

[6] Nadeem Alherbawi, Zarina Shukur and Rossilawati Sulaiman, "A Survey on Data Carving in Digital Forensics",ISSN: 1682-3915 - Asian Journal of Information Technology 15 (24): 5137-5144, 2016.

[7] K. Srinivas, T. Venugopal: Design and Implementation of File Carving Algorithms in Computer Forensics , National Conference on Advances in Computing and Networking, Computer Society of India - JNTUH College of Engineering, Manthani, Karimnagar p81-87, December 2014.

[8] Kulesh Shanmugasundaram, Nasir Memon: *Automatic Reassembly of Document Fragments via Context Based Statistical Models*, ACSAC '03 Proceedings of the 19th Annual Computer Security Applications Conference, IEEE Computer Society Washington, DC, USA.

[9] Xinyan Zha and Sartaj Sahni: *Fast in-place file carving for digital forensics*, Springer Link, 2011.