Multi-version Data recovery for Cluster Identifier Forensics Filesystem with Identifier Integrity

Mohammed Alhussein , Duminda Wijesekera
Department of Computer Science
George Mason University
Fairfax, VA 22030, USA

Abstract— **Recovering deleted information from a hard disk has been a long standing problem. The computer forensics community has addressed information recovery through the development of file carving techniques. Two issues, however, still present significant challenges to their on-going efforts – 1) Prior knowledge of file types is required for building file carvers including file headers and footers, and 2) fragmentation prevents file carvers from successful recovery. As a solution, we propose a forensics file system that embeds a special identifier in every cluster that is either currently allocated or was in the past. The identifier keeps track of every cluster mapping the clusters to a single file irrespective of the file status – existing or deleted. We modified an exFAT implementation on FUSE to implement our forensics file system. We also propose a hashing mechanism that can detect malicious or accidental manipulation of a cluster's identifier. In addition, we introduce the concept of multi-version recovery, where multiple instances of a file can be recovered based on a cluster specific timestamps inserted during the write operation. Finally, using controlled experiments we have been able to verify that our proposed file system successfully recovers all deleted files in our test environment.**

Keywords: computer forensics; data recovery

I. INTRODUCTION

Computer forensics is concerned with computer systems that are involved in crimes. These computer systems can be used to aid in criminal activity. For example, criminals can utilize Internet search engines to obtain information on how to commit a physical crime. Computer systems can also be the target of a crime, such as illegally accessing a system to delete information [1]. In both cases, computer forensics attempts to preserve, collect, recover, analyze and present information from computer systems in a way that is acceptable in court [7].

One significant area of computer forensics is the recovery of evidence. An attacker can deliberately delete information where a deletion can be either the goal of the attack, such as deleting incriminating documents or videos, or the attacker can delete data within the computer system such as log files to hide the attack trail. The data recovery process is also invoked in the case of accidental deletion of data, or in cases where storage devices crash and/or get corrupted. There are many techniques used to recover data from storage devices, depending on the nature of the deleted content and the associated file system. When a user deletes a file, the file system information linking to the deleted file is kept intact, and recovery of the deleted file therefore becomes a straightforward operation in most cases. In the File Allocation Table aka "FAT" file system for example, deleting a file will result in marking the corresponding cluster entries as empty i.e., 0 in the FAT table. However, the information pointing to the actual file will still be present until the corresponding entry in the FAT table is associated with another file. In case where pointer information is no longer available in the FAT table, or if the file system itself is corrupted, data recovery becomes more challenging, requiring more sophisticated techniques in the absence of file meta-data that can lead to the location of the file within the storage unit. Such techniques are referred to as File Carving techniques.

Pal et al define file carving as "a forensics technique that recovers files based merely on file structure and content and without any matching file system meta-data" [2]. Some file carvers use file structure, such as the file header written by the user application, to recover data. Conversely, more advanced file carvers will use knowledge of the file content to recover data employing statistical and/or artificial intelligence techniques. Although file carving is a powerful technique for data recovery, we highlight two issues that present challenges for forensics investigators:

- Prior knowledge of file types and file content
- File fragmentation

In order for a file carver to be built, knowledge of file types and their content is required. Although many current file carvers can handle numerous file types, this is still an issue when faced with new file types. The second and more important problem is fragmentation. Recovering fragmented files present significant challenges to file carving as demonstrated in [3]. As we can see, as the number of fragments increase, the problem becomes much more difficult.

Our system specifically tackles above issues. We propose a forensics file system that can help forensics investigators recover fragmented files without prior knowledge of file types. We introduce the concept of cluster-level identifiers, where the identifying meta-data of a cluster in the form of a file identifier and cluster sequence number are embedded within every cluster. Once a file system is constructed this way, recovering deleted files becomes the process of grouping and ordering disk clusters that belong to different files. We also examine the situation where an attacker is able to gain write access to the volume. We design a mechanism to detect attempts by an attacker to alter the identifier value of one or more clusters to delete, modify or create files. In order to do so, we implemented a cryptographic hashing mechanism where every cluster is hashed, and the hash encrypted by a secret key maintained by the operating system, before the encrypted hash value is stored in the identifier field of the trailing bytes of the cluster.

We also introduce timestamps to the identifier to enable multi-version recovery, where we can recover different instances of a file by grouping and recovering all clusters that have the same file identifier and sequence number using the timestamp value.

The rest of the paper is organized as follows: In section 2 we discuss publications related to the areas of data recovery. In section 3 we describe the proposed system, cluster-level identifiers that facilitate efficient data recovery, identifier integrity and multi-version recovery. In section 4 we describe constrains imposed by the system environment and tools on our design and experimentation. Section 5 details the system design and implementation. In Section 6, we discuss our experimental results. Finally in section 7, we present our conclusions.

II. RELATED WORKS

There has been extensive research into file recovery, and specifically file carving. Carrier, [1] explains general file recovery techniques, while also demonstrating how file systems work and illustrating the file recovery process. Pal et al. [2] lay out the current state of file carving by presenting different file carving techniques. Garfinkle presents the challenges of recovering fragmented files, while pointing out that forensically important files are often fragmented [3]. Outside of file carving, we are only aware of one work that takes a generalized approach to file recovery. Srinivas et al. present a theoretical model where all files are doubly linked lists, by having clusters as the nodes [4]. We are not aware of an implementation of such a solution.

III. PROPOSED SYSTEM

A. Cluster-level identifiers

1) Introduction

A cluster is the smallest unit of storage that is used by the operating system to read and write data. Instead of reading and writing sectors in 512-byte increments, it is more efficient for operating systems to deal with larger blocks of data ranging from 512 bytes to 64 KB, for example in NTFS. Because application data is written by the operating system in blocks with the size of a cluster, it is reasonable to think of forensics, and file recovery specifically in terms of identifying those blocks that comprise the file. As we've discussed in the introduction, however, traditional file recovery techniques require knowledge of the file types to be recovered. Contents of clusters have to be analyzed in order to perform file recovery operations. They also become difficult to apply when the file is fragmented to a few

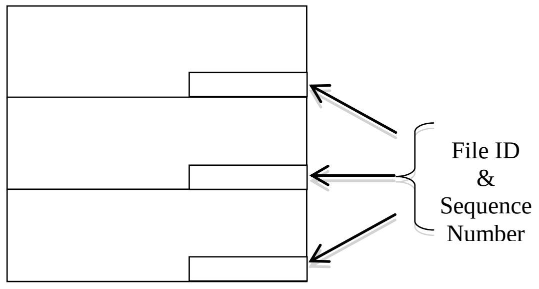

Figure1. Three clusters showing the identifier

fragments, and become unusable when the number of fragments increases. Our forensics file system FFS is designed to work regardless of the number of fragments within a file and without knowledge of file types.

The central concept behind our proposed system is that file identifying information can be embedded within data clusters. In our system, we reserve a number of bytes at the end of every cluster to record the file ID associated with the cluster and the sequence number of the cluster within the group of clusters that comprise the file. By recording the file ID associated with the cluster and the position where the cluster sits within the file, we are able to precisely and efficiently determine where the cluster belongs and in which order.

2) Forensic file system identifier

The identifier stores the file ID and sequence number of the cluster. In order to determine the size needed to store the identifier, we calculate the maximum number of files that can be stored, and the maximum number of clusters that can be in a file. Fig. 1 shows an illustration of the identifier.

Basing our calculations on an implementable version of NTFS, the maximum number of files that can exist in an NTFS volume = $2^{32} – 1$, which requires 4 bytes to store the file ID. Also for a 4 KB cluster system, the maximum number of clusters that we can have per NTFS volume = 2^{32} clusters, which also requires 4 bytes to store the cluster number. Therefore, we will need 8 bytes of storage to store the complete identifier.

B. Environment and considerations

1) Introduction

In order to enable clusters to be embedded with forensic information, changes to the reading/writing mechanism of the operating system must be implemented. Operating systems must ensure that the FFS identifier is embedded into the cluster while performing write operations and removing the FFS identifier correctly when reading data to maintain transparent operations to the applications being served.

2) Kernel v.s. filesystem in userspace

Two approaches were available to us to implement our system. The first was re-implementing the kernel to have the identifier embedded in every cluster by changing the way the reading and writing mechanism operate. The second option was to utilize a kernel module such as FUSE "Filesystem in Userspace" that enables us to run our own customized filesystems without the need to modify the kernel itself [6]. The first approach has the advantage of creating a very efficient implementation, although development will be more complex, requiring considerably more time. Moreover, because recompilation of kernel source code is necessary, only those file systems supported by open source operating systems can be taken under consideration when selecting the file system of choice. We also believe that others can easily adopt a forensics file system running in user space in the forensics community as opposed to kernel-modified operating systems. Therefore, we decided to use the second approach and build our forensics file system on top of FUSE.

3) Filesystem

The criteria we used in selecting a filesystem to customize are as follows:

- Widely adopted filesystem
- Supported by FUSE
- Ease of customization
- Maximum Volume size
- Maximum File size

We looked at a number of filesystems supported by FUSE, such as NTFS, ext3, FAT and exFAT. One issue in particular that was of importance to us was the viability of file system customization without the need to rewrite most of the filesystem from scratch. For example, NTFS-3g provided an NTFS implementation on top of FUSE, but it was not possible to interact with the clusters through the supplied APIs. ExFat was ideal from that point in that the read and write APIs interacted directly with the data clusters. After studying the different file systems we finally settled on exFat as the filesystem of choice.

C. Identifier Integrity

File recovery in our system is dependent on analyzing the file identifier and sequence number in the Identifier field in every cluster to group and order related cluster to reconstruct deleted files. If an attacker however is able to manipulate the value of the Identifier of one or more clusters, then it might be possible for the attacker to change the order of a group of clusters within a file or remove one or more clusters permanently from a file. Therefore maintaining the integrity of the identifier is useful in case an attacker gains access that enable him/her to manipulate the value of the identifier.

To safeguard against such an attack, we calculate the hash value of the identifier after every cluster write operation. We can then proceed to encrypt the resulting hash value using a secret key stored and handled by the underlying operating system. The encrypted hash value is then stored with the identifier associated with the given cluster.

Maintaining the integrity of the identifier can be achieved to an extent by verifying the hash value of the cluster in question. However, this can be costly, because we need to calculate the hash value of the identifier and perform and encryption operation. This approach is based on the assumption that the cryptographic key maintained and secured by the operating system.

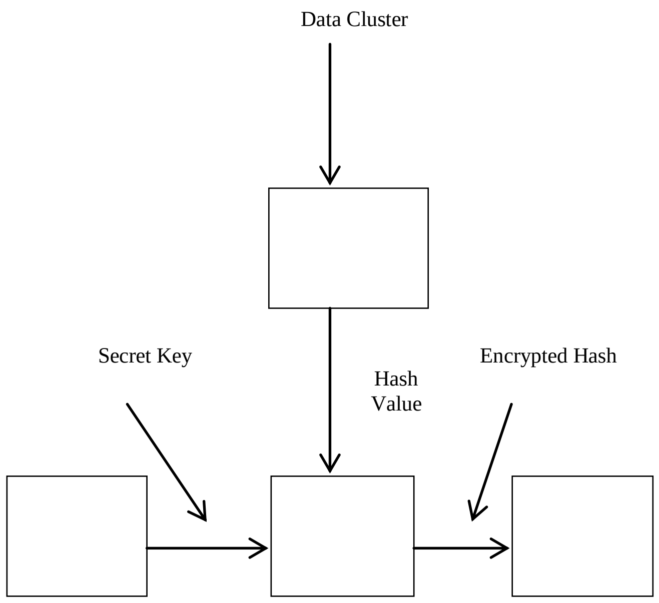

Figure 2. Computing cryptographic hash value of identifier

D. Multiversion Recovery

Standard file operations such as write, append delete, and truncate can affect a file in different ways. More specifically, in our system, updating a file can result in multiple clusters sharing the same file identifier and sequence number. For example, say file A consists of 5 clusters, occupying clusters 201, 202, 203, 204, and 206. Suppose that file B occupies cluster 205. Table 1 shows clusters 201 to 208 with the corresponding files occupying the clusters.

Table 1. Files A and B before updates

201	*file A*	id =1, sq = 1
202	*file A*	id =1, sq = 2
203	*file A*	id=1, sq = 3
204	*file A*	id =1, sq = 4
205	*file B*	id =2, sq = 1
206	*file A*	id =1, sq = 5
207	*unallocated*	
208	*Unallocated*	

Deleting the last cluster from file A, cluster 206 become unoccupied. However, the file ID and sequence number still remains, with values 1 and 5 respectively. Let's assume that file B is deleted, therefore cluster 205 is now unoccupied. If we append a cluster to file A, that cluster will now occupy cluster 205, with file id equal to 1, and sequence number equal to 5. So we have clusters 205 and 206 both with the same file id and sequence numbers. Table 2 shows the state of clusters 201 to 208 after the above operations.

The previous example illustrated the case where we can have multiple clusters with same file id number and the same sequence number. At first, this might create some confusion during the recovery process, because it's not clear which cluster is to selected to include in the recovered file. However, having multiple clusters with sharing the same file id and sequence number can be utilized to recover multiple instances of a file.

In order to be able to recover multiple versions of a file, we introduced a timestamp field to the identifier that can be used to uniquely identify different clusters. We utilize the timestamp to identify the most recent instance of a cluster.

Table 2. Files A and B after the updates

201	*file A*	id =1, sq = 1
202	*file A*	id =1, sq = 2
203	*file A*	id=1, sq = 3
204	*file A*	id =1, sq = 4
205	*file A*	id =1, sq = 5
206	*unallocated*	id =1, sq = 5
207	*unallocated*	
208	*Unallocated*	

IV. Design and Implementation

When an application perform a filesystem operation, the request is passed to the kernel where it is determined if the file is in the FUSE volume and then it is passed to the FUSE kernel module. FUSE then forwards the request to the exFAT Forensics File system, "FFS_exFAT" where the actual operation is being implemented. Fig. 2 describes how applications, kernel, FUSE and FFS_exFAT work together.

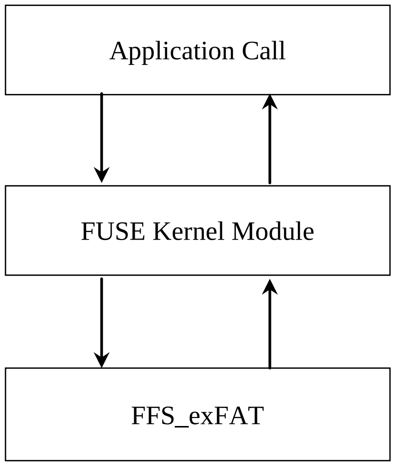

Figure 3. Proposed system architecture

A. *FFS_exFAT*

We utilized the FUSE module exFAT implementation for GNU/Linux developed by Andrew Nayenko [5]. This exFAT implementation on FUSE manipulates files on the cluster level, thus providing a convenient access point to the code to implement the necessary changes that enable the clusters to be embedded with identifiers. We selected a cluster size of 4096 bytes, but the cluster size can be changed to other values.

In order to illustrate the changes to exFAT needed to implement our Forensics Filesystem, we will first explain how exFAT manipulates clusters to perform reading and writing operations. When an application requests to write to file, FUSE passes data in 4K blocks to exFAT. The data is passed through a buffer along with offset from the start of the file and file information. exFAT then writes the information to the appropriate cluster by advancing the cluster pointer in the cluster linked list in *exfat_node* structure. When receives a read operation, exFAT starts reading the data at cluster size increments while advancing the cluster pointer until all data is read.

When designing the Forensics Filesystem, we first need to introduce the concept of "state" to the exFAT implementation. When the system receives a write request, and before applying the FFS Identifier to the cluster, we need to make sure that a mechanism exists to take care of the additional data resulting from removing the last 8 bytes from every cluster to compensate for the addition of the 8-byte identifier. Therefore, a *cluster_buffer* is introduced to the exFAT implementation in order to store data that is to be made available for the following clusters. In the following section, we will explain in detail how both reading and writing and reading routines in exFat have been modified.

1) *Cluster Writing*

The following pseudo code is used for the write operation:

```
fuse_exfat_write(fuse_buffer, block_size);
append_memory(cluster_buffer, fuse_buffer, block_size);
extract_memory(write_buffer, cluster_buffer, block_size - 8);
append_memory(write_buffer, identifier, 8);
exfat_write(write_buffer);
```

Fuse will send a block of data in accordance with the block size described earlier "4KB" and appended to the *cluster_buffer*. The first 4088 byes that result from the combination of the FUSE buffer with cluster buffer will in turn be appended with the 8-byte identifier resulting in 4096 bytes in the exFat write buffer. Depending on the number of bytes sent by FUSE and the number of bytes already in the cluster_buffer, one or more write operations will take place until either the cluster_buffer is empty, or the file is closed. Fig. 3 illustrates the necessary steps for a write operation.

2) *Cluster Reading*

The goal when implementing the read operation is for it to be done in a transparent fashion so that the calling application will receive correct data without the identifier being presented. To do so, we take into account the presence and location of the identifier within the data stream to be read, in addition to the position of data pushed into the current cluster from previous clusters. The following two equations illustrate how the correct range of data is identified from the current and following clusters:

$$x = [(i-1) * 8] \bmod 4088 \quad (1)$$
$$y = [i*8] \quad \bmod 4088 \quad (2)$$

In (1), the value of x is used to determine the starting byte in the current cluster. Data is subsequently read until byte 4088, ignoring the last 8 bytes that make the identifier. Reading continues in the following cluster until we reach the byte specified by y in (2).

For example, when an application attempts to read data form cluster # 115 our system first read cluster # 115, copying data from byte 912 until byte 4088. We then read cluster # 116 and append bytes in the range from 0 until 920 into the reading buffer, ending up with the correct 4096 bytes that represent the cluster to be read. We recall that the last 8 bytes of every cluster read are ignored since they store the value of the identifier, which is irrelevant to the calling application.

V. PERFORMANCE

We previously calculated the need 8 to have bytes to store the identifier. Because we selected 4 KB clusters, to calculate the space within the disk utilized to store the identifier we divide 8 by 4096 = 0.19%.

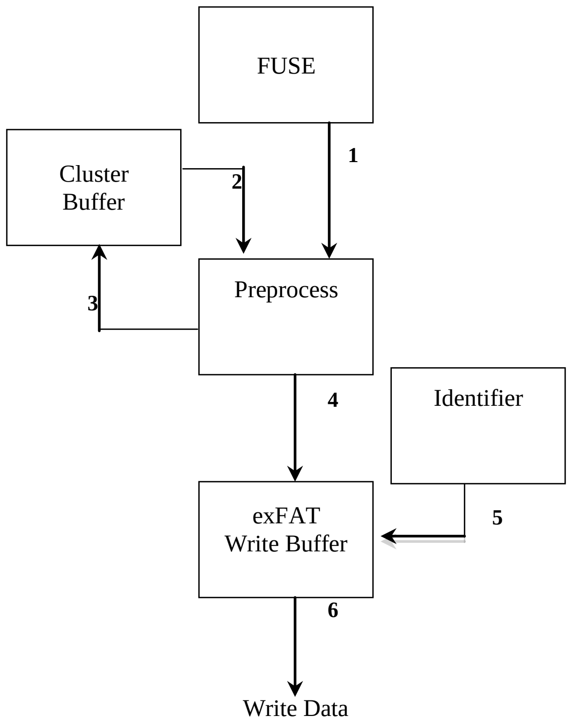

Figure 4. Write routine flowchart

We conclude that by utilizing our system, we are sacrificing less than 0.2% of disk space. Fig. 4 shows how increasing the cluster size can bring down the percentage of space dedicated to the identifier. For example, only 0.012% of disk space is required when cluster size is set to 64 KB.

Our system introduces many memory buffer operations. The major performance penalty to the disk can be correlated specifically to the space complexity. Because we introduce an additional 0.2% of data to be written, we require an additional 0.2% in write operations to write a certain number of clusters. Performance of disk reads follows the same calculation, because we read an additional 0.2% of data in the worse case.

VI. EXPERIMENTAL RESULTS

Ideally, we preferred to test our system on existing hard drives in order to determine how our proposed system would recover files, especially fragmented ones. However, files must be created using our system in the first place utilizing our read/write routines so that the clusters can be embedded with identifiers. We created disk images in a controlled environment simulating user activities resulting in fragmented files.

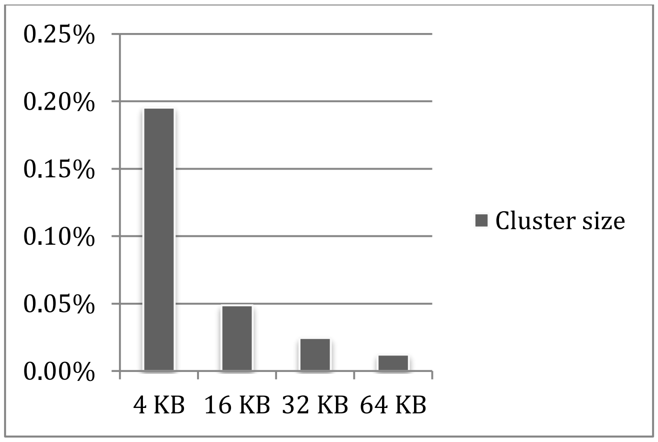

Figure 5. Identifier space overhead as a function of cluster size

We created 4 FFS_exFAT formatted images of size 4GB, and created, copied and deleted files of over 5 MB repeatedly. In our tests, we were able to recover all the deleted files by performing a single pass on the hard drive. We grouped clusters into different pockets according to their file ID value in the identifier, and then order them according to the sequence number.

VII. Conclusion

In this paper, we presented a data recovery approach to recover fragmented files without prior knowledge of file types. Our approach recovers any number of fragments in a file. We built our system by modifying an exFat implementation on top of FUSE. Our initial test shows near perfect results, but our goal is to benchmark against file carving techniques to determine the effectiveness of this system. To do so, as part of our future research, we are building a tool that can transform any existing disk to be embedded with cluster-level identifiers.

References

[1] B. Carrier, File System Forensics Analysis. Adison Wesley Professional , 2005

[2] A. Pal, N. Memon, "The Evolution of File Carving", Signal Processing Magazine, IEEE, Vol 26, issue 2, pp 59-71, March 2009.

[3] S. Garfinkle "Carving Contiguous and Fragmented Files with Fast Object Validation" Proceedings of the 7th annual digital forensics research workshop, Augest 2007

[4] K. Srinivas, T. Bhaskar, "A Novel File System that Facilitates Improved Digital Forensics and Generalized Solution to Fragmented File Recovery", 3rd International Conference on Electronics Computer Technology, India, April 2011.

[5] A. Nayenko, exFAT File System Implemetation – Google Project Hosting. *code.google.com/p/exfat*

[6] FUSE: File System in Userspace. *fuse.sourceforge.net*

[7] Computer Forensics, Vol 8, Number 1, United States Depratment of Justice, January 2008