Using the HFS+ Journal For Deleted File Recovery

By

Aaron Burghardt, Adam Feldman

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2008 USA

Baltimore, MD (Aug 11th - 13th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http:/dfrws.org

available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Using the HFS+ journal for deleted file recovery

*Aaron Burghardt**, *Adam J. Feldman*

Booz Allen Hamilton, Herndon, VA 20171, United States

Keywords:
Mac OS X
HFS+
Journal
Deleted
File
Recovery

ABSTRACT

This paper describes research and analysis that were performed to identify a robust and accurate method for identifying and extracting the residual contents of deleted files stored within an HFS+ file system. A survey performed during 2005 of existing tools and techniques for HFS+ deleted file recovery reinforced the need for newer, more accurate techniques.

Our research and analysis were based on the premise that a transactional history of file I/O operations is maintained in a Journal on HFS+ file systems, and that this history could be used to reconstruct recent deletions of active files from the file system. Such an approach offered a distinct advantage over other current techniques, including recovery of free/unallocated blocks and "file carving" techniques. If the journal entries contained or referenced file attributes such as the extents that specify which file system blocks were occupied by each file, then a much more accurate identification and recovery of deleted file data would be possible.

© 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.

1. Background

Applications for recovering deleted files on Mac OS HFS and HFS+ file systems historically have had limited success compared to recovery tools for other common file systems; the difference is a consequence of HFS's use of B-tree data structures to store metadata that describes the name, block allocation, and other file attributes. When a file is deleted, the B-tree is immediately updated to maintain consistency, which overwrites the file metadata.

With the release of Mac OS X v10.2 in August 2002, Apple enhanced HFS+ by adding metadata journaling, which groups metadata changes into a transactional block. The journaling can be manually enabled or disabled on-the-fly by the user.[1] In version 10.2, journaling was disabled by default. Mac OS X v10.3 was released in October 2003, and it enables the journal by default. Therefore, the recovery technique described here is the most applicable on systems with v10.3 or later installed and on volumes formatted by v10.3 and later systems.

1.1. *The HFS+ file system*[2]

The major components of the HFS+ file system are:

- **Volume header** – contains file system attributes, such as the version and the allocation block size, and information to locate the metadata files.
- **Allocation file** – tracks the usage status of allocation blocks.
- **Catalog file** – contains the majority of file and folder metadata.
- **Extents overflow file** – contains additional extents records for files composed of more fragments than can be recorded in the catalog file.

* *Corresponding author*.
E-mail address: burghardt_aaron@bah.com (A. Burghardt).

[1] With either a command line tool diskutil or the *Disk Utility* application.

[2] A full description of the file system format is provided by Apple in Technote 1150.

1742-2876/$ – see front matter © 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2008.05.013

- **Attributes file** – extensible metadata; it is used for features such as access control lists and Time Machine.
- **Journal file**

The catalog, extents overflow, and attributes files are all instances of a B-tree storage system (Sedgewick, 1990). Only the catalog file is considered in detail here.

1.1.1. *The catalog file*

The catalog file stores catalog file records and catalog folder records, which are the primary store of file and folder metadata, respectively. These records contain:

- Catalog node ID (CNID, a unique unsigned 4-byte integer).
- Timestamps.
- Owner ID.
- Group ID.
- Unix permissions.
- Finder and miscellaneous properties.
- Eight extents records.

An extent record describes the starting block and length in blocks of contiguous blocks that are part of a single fork of a single file (i.e., a single fragment of that file). Thus, the eight extent records in a catalog file record identify the first eight fragments of the file. Additional extents per file are stored in the extents overflow file.[3]

Keys for catalog file and folder records are derived from CNIDs and file names. Each key contains the CNID of the *parent* folder and the name of the file or folder. Using the parent CNID keeps the contents of a folder grouped together in the B-tree nodes.[4]

The catalog file also contains thread records to improve performance when retrieving a file by CNID. A thread record key contains the CNID of the file or folder, but no file name. The data in a thread record is a copy of the key for the file or folder record. Thread records are useful when constructing a path to a file system object; given the key for a file or folder record, the most efficient way to find the key of the parent folder is to retrieve the thread record for the parent CNID. By repeating this process recursively until the root is reached, the full path to the object (relative to the file system's mount point) is obtained.

1.2. *The difficulty of deleted file recovery*

The organization of catalog file data implies that accurate recovery of deleted files can be achieved if the file record and its key can be found. Some existing COTS tools take this approach and employ a scan of the catalog file for deleted file records as the first step. The results of these utilities are often limited to one or two files or no files at all.

This approach is ineffective because the catalog file is not just an index of another data structure; the indexed data is stored within the catalog file. When a file is deleted, the B-tree must be updated to maintain consistency, which may overwrite the deleted file record. In HFS, these updates were performed in a way that occasionally did not erase the record in the B-tree node corresponding to a deleted file. In HFS+, B-tree nodes appear to be updated as a logical unit. In our research, we found that the slack/unused portions of B-tree nodes were consistently filled with 0x00.

1.3. *The journal's role*

An individual update to the file system, from the user's perspective, may result in several related disk updates. Creating a file, for example, may trigger the following changes:

- File and thread records inserted into one catalog file node (which may cascade into several nodes requiring updates).
- The Volume Bitmap file is updated to reflect that the file's content blocks are now in use.
- Records are inserted in the extents overflow if the file is highly fragmented.
- The attributes file is updated if additional attributes are applied.
- The volume header is updated to reflect activity.

All of these updates must be completed or the file system will be corrupted. A power outage or premature removal of an external disk or flash drive is an example where the file system may be interrupted while critical updates are in progress. Journaling was added to HFS+ to address this problem. The steps in a transaction are:

1. Start the transaction by writing a copy all pending file system metadata changes to the journal file.
2. Flush the journal file to disk.
3. Record the presence of the transaction in the journal header.
4. Perform the changes on the actual file system metadata files.
5. Mark the transaction in the journal as completed by updating the journal header.[5]

When a file system is mounted, HFS+ checks the journal for uncommitted transactions. If the transaction did not reach step 3 above, then the changes are lost, but the file system is still consistent. If step 3 completed successfully but step 5 did not, then the entire transaction is replayed, ensuring file system consistency.

1.4. *The journal file*

The journal file is allocated as a contiguous set of blocks on the file system and is never moved or resized. The journal file is implemented as a circular buffer of a fixed size. The beginning of the journal file is the journal header; the remainder of the file is the circular journal buffer. Journaled data is continuously

[3] In practice, only 1–2% of the files of a typical Mac OS X volume will have records in the extents overflow file (Singh, 2006; Singh).

[4] This is a performance optimization for common user tasks, such as displaying or searching the contents of a folder.

[5] There is no need to flush the journal header to disk at this point; disconnecting the disk and leaving the file system dirty will result in a replay of the transaction on the next mount, but that is not incorrect, only redundant.

appended to the file until the end of the buffer is reached. Then, data is written to the beginning of the buffer (skipping over the journal header) again, overwriting old data as it proceeds.

A transaction consists of one or more block lists. A block list consists of a block list header, which describes the blocks in the list, followed by the block data. The block list header is of fixed size. Thus, a transaction may require more than one block list to record all the blocks.

The journal has no inherent knowledge of the file system; it does not contain information about the volume header, catalog file, etc. Rather, it operates at the disk block level: a transaction is simply a collection of disk blocks to be updated. A recovery utility must be able to interpret the blocks for relevant data.

2. Methodology

The deleted file recovery methodology takes advantage of the design and implementation of the journaling to find catalog file records that represent files that have been deleted from the file system. The journal file contains a copy of all of the blocks that are updated in a given transaction, and the contents of the journal file are not erased until the buffer wraps around and overwrites completed transactions. By inference, the journal file will contain copies of catalog file nodes and some of the files within those nodes may no longer exist in the active catalog file (i.e., they are deleted).

2.1. *Accessing the catalog file*

As a prerequisite to implementing this approach, the following subset of HFS+ file system functionality was implemented:

- Given a file record, calculate the location of allocation blocks and read the contents of the file.
- Retrieve B-tree nodes from the catalog file.
- Retrieve records from a B-tree node.
- Perform an HFS+-compliant catalog file key comparison.
- Perform an HFS+-compliant catalog file key search.

2.2. *Employing the HFS+ journal for file recovery*

Once the HFS+ volume and journal concepts are understood, an algorithm for recovering deleted files is straightforward. The implementation of the algorithm included the following steps:

1. Read the volume header.
2. Initialize catalog file access based on information in the volume header.
3. Derive the location of the journal file using the volume header and the journal info block.
4. Read journal file into memory (starting at the oldest transaction, not the beginning of the file).
5. Scan the in-memory copy of the journal file:
   (a) Examine blocks sequentially and identify copies of catalog file B-tree nodes. A node does not have a signature value, but it does have several values that can be checked for consistency and sanity.
   (b) When a catalog file node is identified, iterate over the records stored within. For each record, search the active catalog file for the record. If it isn't found, infer that it is deleted.
   (c) If a record for a deleted file is found, store it in a cache. If the cache contains the record, discard it. (Thus, only the most recent version will be used.)
6. Determine the recoverability potential of deleted files.

2.3. *Determining recovery potential*

The scoring algorithm that we developed is based entirely on the status of the allocation bits in the Volume Bitmap file corresponding to each deleted file. For each block once occupied by a deleted file, a check of the corresponding bit in the Volume Bitmap is performed to ascertain whether it is currently in use by another file. Three possibilities are identified below with a weighted description:

Good. None of the blocks are in use (the contents are probably recoverable).
Partial. Some of the blocks are in use (the value of recovery depends on the file type).
Poor. All of the blocks are in use (and of limited value, though some data may exist in slack space).

For all of these cases, a block that corresponded to a deleted file and is unallocated at the time that the recovery is performed does not necessarily contain the residual data from the deleted file. In fact, the block could have been reused any number of times between the time that the file was deleted and the recovery was performed, and may contain data that is not associated with the referenced file. In all cases, there may still be value in recovering the file blocks so that the slack space (i.e., the data bytes that reside between the logical end of a file and the end of the block in which the end of file is contained) may be examined.

A partially recoverable file is the most complex case and requires the most thorough consideration. The best option depends on the intended use of the data. Our implementation provides three options for files with partial and poor recoverability. They are:

Stop At In-use. This option recovers all blocks that are not in use up to the first in-use block, at which point it stops and does not attempt to copy any later blocks, whether in use or not.
Intact. All of the blocks are recovered as-is, including both allocated and unallocated blocks.
Zeroing. All of the blocks are recovered, but the blocks that are marked as allocated are filled with 0x00. This option is provided so that all unallocated fragments are recovered and confusion from inter-mixed data is avoided.

2.4. *Additional reliability criteria*

Additional analysis could be performed to enhance the accuracy of the recovery potential. For example, transactions in the journal do not have a timestamp, but the transactions

Table 1 – Test Platform: Mac mini

Mac mini	
OS	Mac OS X v10.5
Volumes	disk0s2: Boot volume disk1s2: External firewire disk2s2: External firewire disk5: Disk image
Open applications	Preview TextEdit Terminal Console Xcode Interface Builder Numbers Calculator Safari TeXShop

are chronological and any additional changes to the catalog file are recorded. Therefore, once a deleted file record is identified, it is possible to scan subsequent transactions to determine if any later updates affect the same allocation blocks that the deleted file occupied. If none were found, then the recovery potential could be considered absolute rather than "Good."

3. Empirical analysis

To assess the viability of our technique, some samples were taken to gauge the frequency with which the journal wraps and overwrites historical data. Ultimately, a utility that implements the technique was written and deployed and a sample of the results is provided in the following subsections.

3.1. *Journal wrapping frequency*

A component of our analysis was a risk assessment to address the concern that updates to log files and other file system "chatter" may reduce the window of opportunity and render the technique of little value. To gather data, a small utility polled the active journal start point every 2 s and reported when the journal buffer wrapped. We anticipated that the pattern of usage of the system boot volume would differ significantly from other mounted volumes, so samples were taken of both.

Our test platforms were an Intel Mac mini Core Duo and a MacBook Pro Core Duo, which are described in Tables 1 and 2, respectively. The machines were monitored for about 8 h during which typical user tasks were performed, such as web browsing, reading email, and editing documents. *Mail* on the MacBook Pro was configured with five email accounts, checked email every 5 min, and received and filed approximately 200 emails during the monitored period.

The data was collected and the time interval between each journal wrapping event was calculated. Fig. 1 shows the distribution of the number of journal wrapping events per time interval, with time intervals grouped by the 100. Some key observations include:

- Four volumes were monitored on the Mac mini, but only the boot volume had any occurrences of the journal wrapping.
- The journal file of the Mac mini's boot volume typically wrapped after a 5–10 min interval, but the journal file of the MacBook Pro's boot volume typically wrapped after about a 30-min interval.

Table 2 – Test Platform: MacBook Pro

MacBook Pro	
OS	Mac OS X v10.5
Volumes	disk0s2: Boot volume disk2s2: External firewire (Time Machine enabled)
Open Applications	Mail Preview TextEdit Terminal Console Xcode Interface Builder Calculator Safari Pages Microsoft Word

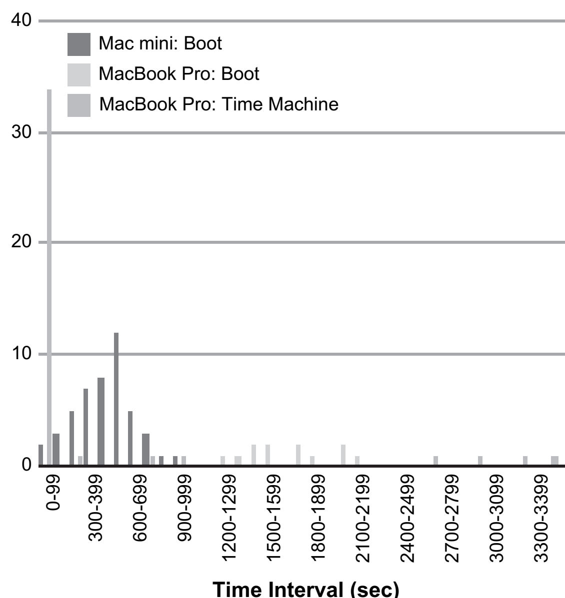

Fig. 1 – Journal wrapping occurrences per time interval.

Table 3 – Recovery Potential Analysis Summary

Volume	Good	Partial	Poor	Total
MBP: disk0s2	59	0	8	67
MBP: disk2s2	3	0	0	3
Mini: disk0s2	10	0	4	14
Mini: disk1s2	32	0	87	119
Mini: disk2s2	14	0	22	36
Mini: disk5	141	0	21	162

Table 4 – File recovery screenshot

File name	Potential	Data size	Rsrc size	CNID	Create date	Modification date	Access date	Path
index	Good	1.177	0	282122	2008-03-12, 12:38	2008-03-12, 12.38	2008-03-12, 12.38	/Documents/DFRWS/.git/index
master	Good	41	0	280895	2008-03-10, 04:15	2008-03-10, 04:15	2008-03-12, 13:54	/Documents/DFRWS/.git/refs/heads/master
index	Good	1177	0	282123	2008-03-12, 12:41	2008-03-12, 12:41	2008-03-12, 12:41	/Documents/DFRWS/.git/index
Journal Recovery.text	Good	23,733	0	282125	2008-03-12, 13:17	2008-03-12, 13:17	2008-03-12, 13:17	/Documents/DFRWS/Journal Recovery.tex
Journal Recovery.text	Partial	23,733	0	282126	2008-03-12, 13:18	2008-03-12, 13:18	2008-03-12, 13:18	/Documents/DFRWS/Journal Recovery.tex
Journal Recovery.text	Poor	23,733	0	282127	2008-03-12, 13:19	2008-03-12, 13:19	2008-03-12, 13:19	/Documents/DFRWS/Journal Recovery.tex
Journal Recovery.text	Partial	21,119	0	282129	2008-03-12, 13:19	2008-03-12, 13:24	2008-03-12, 13:24	/Documents/DFRWS/Journal Recovery.tex
index	Good	1177	0	282132	2008-03-12, 13:54	2008-03-12, 13:54	2008-03-12, 13:54	/Documents/DFRWS/.git/index
About BDAlias+wolf.	Good	737	0	278879	2007-04-08, 2:56	2007-04-08, 02:56	2008-03-04, 16:35	/Orphanded/About BDAlias +wolf.rtf
About BDAlias.rtf	Good	4134	0	278880	2007-04-08, 02:55	2007-04-08, 02:55	2008-03-04, 16:35	/Orphanded/About BDAlias.rtf
user.sty	Good	3204	0	280811	2008-02-03, 17:21	2008-02-07, 18:37	2008-03-09, 21:46	/Orphanded/user.sty
BDAlias.h	Good	3712	0	278881	2007-04-08, 02:55	2007-08-27, 00:12	2008-03-10, 04:21	/Orphanded/BDAlias.h
BDAlias.m	Good	15,183	0	279136	2007-04-09, 14:22	2008-02-25, 10:40	2008-03-10, 04:21	/Orphanded/BDAlias.m
BDAlias.rtf	Good	6095	0	278883	2007-04-08, 02:55	2007-04-08, 02:55	2008-03-04, 16:35	/Orphanded/BDAlias.rtf
DFRWS Outline.doc	Good	25,088	0	280812	2008-01-08, 15:44	2008-01-08, 15:44	2008-03-09, 21:46	Orphanded/DFRWS Outline.doc
user.mode1v3	Good	47,775	0	281945	2008-03-10, 23:27	2008-03-10, 23:27	2008-03-10, 23:27	/Myproject/MyUtility/Uber Utility.xcodeproj/user.mode1v3
user.pbxuser	Good	40,503	0	281947	2008-03-10, 23:27	2008-03-10, 23:27	2008-03-10, 23:27	/Myproject/MyUtility/Uber Utility.xcodeproj/user.pbxuser
project.pbxproj	Good	29,111	0	281946	2008-03-10, 23:27	2008-03-10, 23:27	2008-03-10, 23:27	/Myproject/MyUtility/Uber.Utility.xcodeproj/project.pbxproj
DisklmageExporter.m	Good	5225	0	281949	2008-03-10, 10:54	2008-03-10, 23:32	2008-03-10, 23:32	/Myproject/MyUtility/Source/DisklmageExporter.m
index	Good	1177	0	280874	2008-03-10, 04:15	2008-03-10, 04:15	2008-03-10, 04:15	/Documents/DFRWS/.git/index
classes.nib	Good	772	0	281794	2008-03-10, 13:43	2008-03-10, 13:43	2008-03-10, 23:48	orphaned/classes.nib
info.nib	Good	582	0	281795	2008-03-10, 13:43	2008-03-10, 13:43	2008-03-10, 23:48	Orphaned/info.nib
Keyedobjects.nib	Good	19,215	0	281796	2008-03-10, 13:43	2008-03-10, 13:43	2008-03-10, 23:48	Orphaned/Keyedobjects.nib
user.mode1v3	Good	47,161	0	281901	2008-03-10, 16:46	2008-03-10, 16:46	2008-03-10, 16:46	/Myproject/MyUtility/Uber Utility.xcodeproj/user.mode1v3
user.pbxuser	Good	61,913	0	281903	2008-03-10, 16:46	2008-03-10, 16:46	2008-03-10, 16:46	/Myproject/MyUtility/Uber Utility.xcodeproj/user.pbxuser
project.pbxproj	Good	29,111	0	281902	2008-03-10, 16:46	2008-03-10, 16:46	2008-03-10, 16:46	/Myproject/MyUtility/Uber Utility.xcodeproj/project.pbxproj
user.mode1v3	Good	44,311	0	281904	2008-03-10, 16:46	2008-03-10, 16:46	2008-03-10, 16:46	/Myproject/Research Tools/Research Tools.xcodeproj/user.mode1v3
userpbxuser	Good	20,759	0	281906	2008-03-10, 16:46	2008-03-10, 16:46	2008-03-10, 16:46	/Myproject/Research Tools/Research Tools.xcodeproj/user.pbxuser
project.pbxproj	Good	27,180	0	281905	2008-03-10, 16:46	2008-03-10, 16:46	2008-03-10, 16:46	/Myproject/Research Tools/Research Tools.xcodeproj/project.pbxproj
SpareBlock.m	Good	461	0	281908	2008-02-20, 05:15	2008-03-10, 16:54	2008-03-10, 16:46	/Myproject/MyUtility/Source/SpareBlock.m
MyDocument.h	Good	421	0	281784	2008-02-19, 15:20	2008-03-10, 13:41	2008-03-10, 23:59	/Myproject/MyUtility/Source/MyDocument.h
MyDocument.m	Good	3883	0	281803	2008-02-19, 15:20	2008-03-10, 13:43	2008-03-11, 00:20	/Myproject/MyUtility/Source/MyDocument.m
MyDocumentWindow	Good	725	0	281786	2008-02-23, 21:10	2008-03-10, 13:41	2008-03-11, 00:35	/Myproject/MyUtility/Source/MyDocumentwindowController.h
MyDocumentWindow	Good	6340	0	281788	2008-02-23, 21:10	2008-03-10, 13:41	2008-03-11, 00:20	/Myproject/MyUtility/Source/MyDocumentwindowController.m
DisklmageExporter.m	Good	5204	0	281892	2008-03-10, 10:54	2008-03-10, 16:36	2008-03-10, 23:31	/Myproject/MyUtility/Source/DisklmageExporter.m
Parser.h	Good	936	0	280602	2008-02-19, 15:48	2008-03-05, 05:40	2008-03-11, 08:50	/Myproject/MyUtility/Source/Parser.h
user.pbxuser	Good	62,264	0	281910	2008-03-10, 16:55	2008-03-10, 16:55	2008-03-10, 16:55	/Myproject/MyUtility/Uber Utility.xcodeproj/user.pbxuser

- The journal file on the Time Machine volume of the MacBook Pro regularly wrapped in 20–60 s, but never less than 20 s, while a backup was in process. When a backup was not in progress, it did not wrap.

3.2. *Recovery potential analysis*

Our implementation of this journal-based method was used to analyze the volumes described in Section 3.1. A summary of these results is presented in Table 3. In our research, we observed that files with a partial recovery potential were uncommon; in the results presented in Table 3, none were identified.

3.3. *A real-world example*

Table 4 was generated with data from a utility that implements our technique. The file system is a flash drive used by the author during preparation of this paper. In this case, 120 instances of deleted files were found in the journal file, though only a subset of data is shown. The table illustrates the detailed metadata that is recovered using this approach.

3.4. *Limitations*

Using the journal file for recovering deleted files as described in this paper has inherent limitations:

- The journal file is a circular buffer, so the history is limited by the frequency at which the buffer wraps. In our experience, this time-frame can range from a few minutes on an active boot volume to several hours on secondary volumes.
- The potential to recover a deleted file – even one that is recently deleted – is not guaranteed. When a catalog file node is updated after a file is deleted, an old copy of the node is not written to the journal file; only the new version of a node is written, and the new version does not contain the file record. Thus, previous file system activity must have occurred to cause the file record to be written to the journal prior to deletion. Opening a file or moving it to the trash may trigger this activity, as can an update on a different file whose record happens to be in the same B-tree node.
- The full path of the file may not be retrievable. The path is constructed recursively, so if any of the file's parent folders has been deleted, attempting to reconstruct the path may fail, in which case the file is orphaned.
- Shadow files may mask potential results. A common approach to Mac forensics is to use dd to make an image of a disk. Next, hdiutil is used with the shadow option to attach the image, which creates /dev/disk* and /dev/rdisk* device entries, and mount the file systems in the image. The shadow file protects the original image file from modification by allowing write access to the attached device, but the writes are recorded in a separate file rather than the original image file. On subsequent reads, if the block to be read is in the shadow file, the block from the shadow file is returned rather than the original block from the image file. This is done without the file system's knowledge and allows it to mount file systems that it otherwise would not.

  When a shadow file is used, however, the /dev entries are available as read/write devices, not write-protected or read-only (the *image file* is read-only, but not the device entry). If the record for a deleted file is in a section of the journal file that is overwritten when the file system is mounted, the file record will no longer be recoverable because the file recovery implementation will see the updated version of the journal file, not what is in the disk image. Therefore, if a recovery utility uses the /dev/disk* or /dev/rdisk* device to access an attached disk image, the disk should be attached without a shadow file and without mounting the file system.
- The time when a file was deleted is not known.
- Fragmented files which have extents records in the extents overflow file impose an additional obstacle. The extent records (i.e., fragments) stored in the extents overflow are managed separately from the catalog file records, so it is possible that a deleted catalog file record is found, but associated extents overflow records are not found. In this scenario, only the first eight fragments can be recovered.
- While an allocation block may be free at the time of recovery, it may have been allocated and unallocated a number of times between the time the corresponding file was deleted and recovered.

4. Summary

Our research has identified a viable method to detect and recover deleted files on HFS+ file systems that have journaling enabled, and provides a useful complement to established techniques and tools. Recovery of the file contents can be performed with more accuracy than other file carving techniques because the exact range of allocation blocks is known. The method is successful even if the allocation blocks are separated into multiple fragments – a situation that many other techniques do not take into account. Empirical results of our testing consistently identify dozens of potential files for recovery. Additionally, for each file recovered, important metadata like the file name and timestamps are also recovered.

An important limitation of our approach is the limited duration of recorded transactions within the journal file (i.e., the journal's circular buffer frequently wraps, overwriting its oldest history). Moreover, not all deleted files will be recoverable because the technique is dependent on another independent file system event to write a reference to the file into the journal file.

REFERENCES

Apple. Technical note TN1150: HFS plus volume format. http://developer.apple.com/technotes/tn/tn1150.html.

Sedgewick Robert. Algorithms in C. Addison-Wesley Professional; January 1990.
Singh Amit. hfsdebug: a debugger for hfs plus volumes. http://www.kernelthread.com/software/hfsdebug.
Singh Amit. Mac Os X internals: a systems approach. Addison-Wesley Professional; June 2006.

Aaron Burghardt is employed as a consultant for Booz Allen Hamilton. His primary focus has been development of specialized Macintosh forensic tools.

Adam Feldman has 22 years of professional and engineering services and management experience in the areas of information and computer security, software engineering, and investigative technologies - including computer forensics and digital data and text analysis. Currently, he provides technical and strategic direction, business development, and thought leadership for several investigative technologies and data analytics initiatives, as well as program management for computer forensics and text analytics projects for federal government clients.