Byteprints: A Tool to Gather Digital Evidence

Sriranjani Sitaraman, Srinivasan Krishnamurthy and S. Venkatesan
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688.
{ginss, ksrini}@student.utdallas.edu, venky@utdallas.edu

Abstract

In this paper, we present techniques to recover useful information from disk drives that are used to store user data. The main idea is to use a logging mechanism to record the modifications to each disk block, and then employ fast algorithms to reconstruct the contents of a file (or a directory) as it existed sometime in the past. Such a consistent snapshot of a file may be used to determine whether a given file ever existed on disk, to undelete a file that was deleted long ago, or to obtain a timeline of activities on a file. This can also be used to validate that a file with given contents existed at some time in the past or to refute a claim that a file existed in a time interval. Information gathered using these consistent snapshots can be used as valuable digital evidence.

Keywords: Digital Forensics, Digital Evidence, Consistent Snapshot, Checkpointing, Rollback

1. Introduction

Magnetic storage media like hard disks have become central to our means of storing and processing data. Data in a hard disk can be modified or deleted. Unix commands such as *rm*, *mkfs*, *format*, etc., do not actually delete most of the data; they just change the definition of the file system for the operating system that performs data reads and writes [11]. Useful data can be found in unallocated disk blocks and in unused partitions of the hard disk. Also, when a file is deleted, the contents of the file are not immediately erased and overwritten. Instead, the blocks that contained the file's contents are added to the free list. Thus, by examining all the blocks, parts of the deleted file or the entire deleted file can be reconstructed, but this step is very tedious and time consuming.

There is abundant literature to date about data remaining on magnetic media and how it can be recovered [2] [6] [7] [14]. Casey [3] discusses various approaches to recover encrypted digital evidence. Documents like TLDP's Linux Ext2fs Undeletion mini-HOWTO [4], ext2fs/debugfs [8] provide information about how to find data that is otherwise hidden to the file system. Open source utilities like *unrm* and *lazarus* found in The Coroner's Toolkit [5] have been developed for recovering deleted Unix files.

The above-mentioned utilities take advantage of the availability of remnants of deleted files in various sectors of the hard disk to recover useful information. However, these techniques are not very effective when the data in the hard disk is overwritten. Overwriting data may be viewed as creating a new "layer" of bytes over the existing layer of data in the hard disk. We refer to these layers of data on magnetic medium as *'Byteprints'*. Typically, we can access only the top layer. The byteprints can be obtained using effective logging techniques wherein every change to each data block is recorded. A change to a data block with address b that is written to disk at time t can be logged remotely as a 3-tuple (b, new contents of b, t). Note that except for video and certain other large files, user-created files are fairly small in size and changes to their contents usually affect only a small number of data blocks. Therefore, all changes to user files can be captured by the logging mechanism without much overhead.

Advanced techniques like Magnetic Force Microscopy (MFM) and Scanning Tunneling Microscopy (STM), which image magnetized patterns, can also be used to recover overwritten data [9] [10] [13] [15]. These techniques exploit the fact that it is virtually impossible to write data to the exact same location at every instance because of physical limitations of the recording mechanisms. However, we are not aware of these MFM-like devices being used commercially for data recovery, and they incur huge costs in time and storage space. Further, as the magnetic medium is overwritten successively, it becomes increasingly difficult to obtain lower layers of the data remnants [12].

In this paper, we present a method to obtain snapshots of modified or deleted files as they existed on disk at different time instants in the past. An application of obtaining such

IEEE COMPUTER SOCIETY

snapshots of a file is determining if a given file, say */a/b/c*, ever existed in the hard disk even if the magnetic medium had been deliberately overwritten multiple times to hide its existence. Our method can also be used to reconstruct a given file even if it had been deleted and its data blocks had been re-written many times. An obvious use of a tool with such a capability is in gathering digital evidence that may be used to prosecute cyber-criminals.

2. System Model

We assume that a Unix-based file system is used in the disk drive. Even though we have focussed on the Second Extended file system (Ext2), the ideas presented in this work can be easily adopted to various other file systems. We assume the availability of a logging mechanism or devices like MFM for retrieving byteprints. The logging mechanism can be implemented by trapping the writes to the physical disk and sending a copy of the block along with the time and the block address to a remote host.

The components of a file in any file system are its data and the meta-data that contains information about how the file's data is organized. Specifically, in the second extended file system (Ext2), the components of a file are its inode (meta-data) and data. The *inode* contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times [1]. In order to obtain the data, we need to know the addresses of the data blocks that have the contents of the file. These data block addresses can be found in the inode and possibly some indirect data blocks.

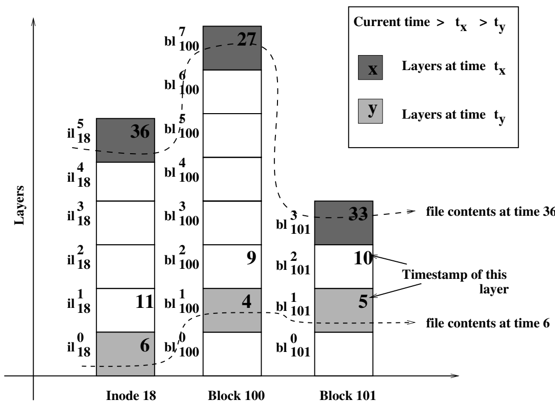

Figure 1. Byteprint Stacks.

Consider block *b* of the hard disk that has been overwritten multiple times, in effect creating a stack of byteprints for this block. The byteprint stacks of different data blocks may have varying height. For example, as shown in Figure 1, data block 101 has been written fewer number of times than data block 100. The number inside each layer in Figure 1 indicates the timestamp of the layer. For example, the timestamp of the topmost layer of block 100 is 27. Let the layers of block *i* be named bl_i^0, bl_i^1, etc., where the subscript refers to the block number and the superscript refers to the number of the byteprint layer. Note that bl_i^0 is the first (bottommost) layer (or layer 0) created for block *i* on the disk. At any time, two data blocks may have different number of byteprint layers. In addition to data blocks, byteprints for inodes are also possible. Every time an inode is modified, a new layer is created for the disk block in which the modified inode is stored. We refer to the mth layer of the data block that stores inode *i* as il_i^m. In Figure 1, the topmost layer of the block in which inode 18 resides is referred to as il_{18}^5. The current state of the disk consists of the topmost layers in the byteprint stacks of all the data blocks, shown in Figure 1 as the layers corresponding to time t_x.

2.1. Definitions

The following terms are used subsequently.

***BS*[*b*, *t*]:** For a given block number *b* and time *t*, the layer of byteprints of block *b* that was created at the latest time t_1 such that $t_1 \leq t$ is referred to as its *Block State*, *BS*[*b*, *t*]. The contents of block *b* at time *t* are found in the byteprint layer *BS*[*b*, *t*].

***IS*[*i*, *t*]:** For a given inode *i* and time *t*, *Inode State*, *IS*[*i*, *t*], is the layer of inode *i* that was created at the latest time t_1 such that $t_1 \leq t$. The contents of the inode *i* at time *t* are in the inode layer *IS*[*i*, *t*].

Snapshot of a File: We define the *snapshot* of a file *f* at time *t* to consist of the layer of the inode of *f* and the corresponding layer of each of its data blocks (direct and indirect) that existed at time *t*. Consider a file */a/b*, associated with inode 18, with all of its data stored in data blocks 100 and 101. The snapshot of the file */a/b* at time t_y (say t_y = 6) consists of layers il_{18}^0, bl_{100}^1 and bl_{101}^1 as shown in Figure 1. Note that each data block may have been overwritten independent of each other, and hence, a valid file may consist of mth layer of block *i* and n^{th} layer of block j and $m \neq n$.

Consistent Snapshot of a File: We refer to a snapshot of a file *f* at time *t*, denoted by *consistent_snap*(*f*, *t*), as being *consistent* if it represents the copy of the file *f* as viewed by a user via the file system interface at time *t*. In other words, the consistent snapshot of a file includes all disk writes pertaining to every modification done to file *f* (its inode and data blocks) until time *t*. Clearly, *consistent_snap*(*f*, *t*) is the consistent snapshot of *f* that was created at the latest time t_1 such that $t_1 \leq t$.

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0-7695-2315-3/05 $ 20.00 **IEEE**

IEEE COMPUTER SOCIETY

We next present the various assumptions that are needed for our work.

2.2. Assumptions

The following are assumed about the policies regarding disk writes and the order in which the writes are performed. These policies are assumed to be adopted by the file system layer (of the OS) that is responsible for creating, deleting or modifying files.

- When a file is modified, the modified data blocks are first written to the disk and then the file's inode are written to disk.
- When a file is created, the data blocks of the file, the inode of the file, the data blocks of its parent directory, and the inode of its parent directory are written to disk in that order.
- When deleting a file, the following is the order of disk writes: (1) the modified data block of the parent directory and the modified inode of the parent directory are first written to disk, (2) the deleted file's modified inode is written (and if possible, the deleted inode is released to the free list of inodes), and (3) the data blocks of the deleted files are added to the free list.

The ordering of disk writes can be achieved using synchronous write operations where the process waits for the completion of the write operation before performing any other action. In this paper, we assume that an intruder's actions cannot change the sequence of disk writes. The particular ordering of disk writes, as mentioned in the above policies, is essential for avoiding inconsistent disk states in case of a system crash during the write process [1]. For example, when a file is deleted, the file name is removed from the parent directory and the updated directory is written synchronously to the disk before the contents of the file itself are destroyed and the inode of the file is released to the free list. If the system crashes before the file contents are removed, damage to the file system is small. There would be an inode with a link count that is one greater than the number of directory entries pointing to it, but all other paths to the file would still be valid. If the directory write is not synchronous or if the order of the writes is reversed, it is possible for the directory entry to point to a free or reallocated inode after a system crash. In this case, there would be more directory entries pointing to an inode than indicated by the link count in the inode. As a result, greater effort would be required to clean the file system in the latter case.

With a mechanism to obtain a consistent snapshot of a file at a time instant t, a number of problems can be solved.

1. Checking whether a particular file ever existed on disk.
2. Undeleting a given file or a directory; obtaining a consistent snapshot of a file at a particular time.
3. Performing a timeline analysis of the events that occurred prior to a time instant t.

In Section 4, we describe an algorithm that can be used to solve one of the problems listed above, namely, the problem of finding whether a particular file ever existed in a given directory in the file system tree. With a technique to obtain the consistent snapshot of a file at a given time instant, a file can be undeleted by rolling back to the latest consistent snapshot of the file. Having a timeline of file activity can help identify areas of a file system that may contain digital evidence. Reconstruction of system events can be used in computer forensic tools to determine what happened in the system prior to a particular time (say, the time when an intrusion was detected). This understanding of computer intrusion patterns can be used to find exploited vulnerabilities in system programs, to defend against future intrusions, etc.

3. Consistent Snapshot of a File

In this section, we describe how we can obtain *consistent_snap*(f, t) of a file f at time t.

3.1. Need for Layer Timestamps

Let i_f be the inode of the file f under consideration. The byteprint layer of the inode i_f of file f at a particular time instant t can be obtained by repeatedly retrieving the byteprint layers from the top to the bottom of the stack and scanning the contents of each layer for the modification time of inode i_f. $IS[i, t]$ is the first layer in the inode's byteprint stack that has a modification time that is less than or equal to time t.

After obtaining $IS[i, t]$, we can obtain the addresses of the direct data blocks that contain the contents of file f by examining the inode i_f. If f is a long file, then we have to scan the contents of single, double or triple indirect blocks to obtain the direct block addresses. A single indirect block refers to a block that contains a list of direct block numbers [1]. The double indirect block contains a list of indirect block numbers, and the triple indirect block contains a list of double indirect block numbers as shown in Figure 2. Let us consider the case of single indirect blocks. To access the data via the single indirect block, the kernel must read the indirect block, find the appropriate direct block addresses, and then read the direct blocks to find the data. From the inode i_f, we can only obtain the address of the single indirect block, but in order to get the addresses of the direct blocks that contain data, we need to find which layer of the single indirect block is consistent with $IS[i, t]$. An inode of a file stores only the addresses of the disk blocks where the

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0-7695-2315-3/05 $ 20.00 **IEEE**

IEEE COMPUTER SOCIETY

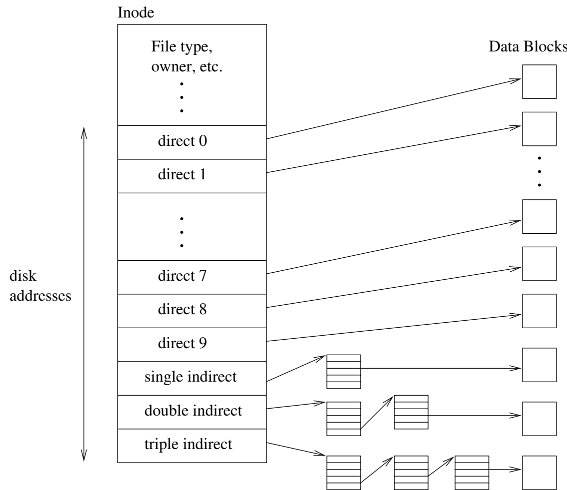

Figure 2. Direct and Indirect Blocks in an Inode.

file's contents are stored and does not store details about the times at which different byteprint layers were created for the disk blocks associated with it. For the indirect block b, we cannot determine $BS[b, t]$ because neither the contents of the inode at time t nor the contents of the different layers of the indirect block contain information about $BS[b, t]$. Similarly, even after obtaining a direct block address that contains the file data, we cannot determine which byteprint layer is part of the consistent snapshot of file f that existed at time t. Thus, it may not be possible to obtain a consistent snapshot of a file completely if the timestamps associated with each byteprint layer are not available.

We next examine how we can determine if a file f existed on the disk at time t by the brute force approach of examining all possible combinations of byteprint layers of various data blocks of the hard disk. In order to apply the brute force method, we assume the following:

- The contents of the file f as it existed at time t are known.
- There exists a mechanism to retrieve all layers of byteprints of any block.
- f is a small file, and its contents are stored in a few direct blocks whose addresses can be obtained within the inode i_f itself. Indirect blocks need not be considered in this case.

Suppose that after obtaining the addresses of the direct blocks, we attempt to examine all possible combinations of the different byteprint layers of the direct blocks. Let one such combination contain data consistent with the known contents of the file. Such a combination of byteprint layers of the direct data blocks is only a candidate for being the snapshot of file f. The uncertainty arises because in the current file system structure, the time of creation of each layer of byteprints is not stored anywhere. We know that the data blocks are not restricted to be associated with only one inode, i.e. i_f. Suppose that these direct blocks were associated with some other file f_1 instead of f at some time t_1, where $t_1 \neq t$, and contained exactly the same data. Given this scenario, we cannot claim that the byteprint layers of the direct blocks that were obtained belonged to file f at time t. Thus, we may not be able to obtain the correct consistent snapshot of a file by considering all possible combinations of layers of byteprints, even if we are allowed infinite time and resources to conduct such an experiment. Hence, in order to obtain *consistent_snap*(f, t), it appears that we need some mechanism that would timestamp the various byteprint layers.

As noted earlier, every time a data block of a file is modified, the address of the data block, the new contents of the block and the time at which the modified block is written to the hard disk can be logged. Using this logging mechanism, we can easily obtain the timestamp of each layer. On the other hand, if we were to use MFM-like devices, we can store the timestamp in the data block itself and update it whenever a new layer is created.

3.2. Obtaining a Consistent Snapshot of a File

Assuming we have access to various byteprint layers and their timestamps, we now describe an algorithm to obtain *consistent_snap*(f, t) of a file f with inode i_f at time t. Note that f, i_f and t are given as input to the problem. The following two points are helpful.

1. For the inode, we choose $IS[i_f, t]$ to be part of the consistent snapshot. Let the timestamp (creation time) of $IS[i_f, t]$ be t_1.
2. For each block b_i listed in $IS[i_f, t]$, we choose $BS[b_i, t_1]$ to be part of the consistent snapshot.

The pseudocode of the algorithm follows:

Algorithm: *consistent_snap*(f, t)

add $IS[i_f, t]$ to *consistent_snap*(f, t) /* *Initialization* */
t_1 = time stamp of $IS[i_f, t]$
for each block address b_i (direct or indirect) found in layer $IS[i_f, t]$ of i_f {
  $BS[b_i, t_1]$ = the layer of block b_i that existed at time t_1
  add $BS[b_i, t_1]$ to *consistent_snap*(f, t)
}

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0-7695-2315-3/05 $ 20.00 **IEEE**

IEEE COMPUTER SOCIETY

We now provide a proof that the snapshot obtained in this manner is consistent.

Theorem 1 *The snapshot, consistent_snap(f, t), of a file f at time t as described above is consistent.*

Proof: In the initial state of the disk, the data blocks have never been written to, and hence, there would be no byteprint layers created on disk. The only possible operations on files that will cause disk writes are the creation, deletion and modification of files. The assumptions about the order of synchronous disk writes as mentioned in Section 2 ensure that the timestamp of the inode layer created is greater than the timestamp of the layer created for any modified data block of the file during any creation, deletion or modification operation performed on the file. Thus, by first determining the layer of the file's inode that existed at time *t* and then finding the data block layers consistent with this layer of the inode, we ensure that we retrieve *consistent_snap*(*f*, *t*) correctly. ■

4. An Application

In this section, we describe an algorithm that can be used to determine whether a file, say */a/b/c*, ever existed on disk and, (1) if it existed, we can get a version of the file contents, and, (2) if the file contents are known, verify if the contents of the file obtained match the known contents. The basic idea is to determine the time interval in which the file */a/b/c* existed on disk, and then verify the contents of all possible consistent snapshots of */a/b/c* found in that time interval. We obtain the time interval within which the file */a/b/c* existed on disk by searching all consistent snapshots of the parent directory */a/b* for an entry for file *c*.

For ease of exposition, we make the following assumptions about the file */a/b/c* as we are only illustrating a use of obtaining *consistent_snap*(*f*, *t*) of a file *f* at time *t*.

1. Parent directory, */a/b*, existed from the initial state of the disk and was not deleted subsequently.

2. Some information about file */a/b/c* such as its contents, type or format is known.

3. File */a/b/c* was not renamed.

Note that our algorithm can be suitably modified to accommodate the lack of any or all of the above assumptions.

We present the details of the algorithm in the form of two subroutines in this section. Procedure *main()* takes the pathname of the file to be searched as input and returns the result of its search. After obtaining a consistent snapshot of the file, procedure *main()* invokes subroutine *verify_contents()* and gets the file contents validated.

Procedure: main()

Input: File being searched: */a/b/c*
Output: 1, file is found (or) 0, file not found

```
m = current layer of inode of parent directory, /a/b
t = current timestamp
while (m ≠ null) {
   t1 = Timestamp of m
   for each data block address b found in layer m of inode
   of parent directory {
      /* get the layer of block b consistent with timestamp
      t1 */
      i = BS[b, t1]
      Check for file name entry c in layer i of block b
      if entry for file name c is found with associated
      non-zero inode number, ic {
         /* verify contents of all possible consistent
         snapshots of file found in time interval [t .. t1] */
         result = verify_contents(ic, t, t1);
         if (result = 1) return 1;
      }
   }
   m = predecessor(m); /* predecessor(m) is the byteprint
      layer created immediately before the m^th layer*/
   t = t1;
}
/* checked all layers of parent inode, file not found */
return 0;
```

Procedure: verify_contents(inode i_c, time t_1, time t_2)

Possible data validity conditions include:

- If contents of file are known, then candidate file contents should match exactly.

- If some information about the file is known, such as the type, structure, or format, etc., then the candidate file contents should indicate a similar file.

- If a hash value of the file is given, then the candidate file contents should hash to the same value.

Input:
i_c = Inode number of file to be verified
$[t_1, t_2]$ = Time interval between which inode i_c was associated with the filename */a/b/c*

Output: 1, if contents satisfy data validity conditions, 0 otherwise

```
m = current layer of inode ic
while (m ≠ null) {
   t = timestamp of layer m of ic
```


Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0-7695-2315-3/05 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

```
    if (t1 ≥ t ≥ t2) {
        Find consistent snapshot of file c at time t
        if (contents of snapshot satisfy the data validity
        conditions)
            return 1;
    }
    m = predecessor(m); /* check in previous layer of inode
        ic */
}
return 0;
```

Time Complexity: Let the number of blocks associated with a consistent snapshot of a file be *N*. Let the disk block size be *B* bytes. Let x be the number of layers in the byteprint stack of the inode of the parent directory */a/b* and *y* be the number of layers in the byteprint stack of the inode of the file */a/b/c*. In the worst case, we may examine the consistent snapshot of the parent directory file */a/b* corresponding to all x layers of the inode of */a/b* for an entry for file *c*. This may in turn result in a time interval that causes all the *y* layers of the byteprint stack of the inode of file */a/b/c* and their corresponding consistent snapshots of */a/b/c* to be examined. Thus, the worst case time complexity can be calculated as T = $O(NB(x + y))$.

Space Overhead: To estimate the storage space requirement for the proposed logging mechanism, we observed the rate of disk writes performed in five computers [1] in our university network. An average rate of disk writes of 24KB per second over all computers was observed. The maximum recorded rate of disk writes was 41KB per second. Based on the average disk write rate, we would require about 2GB of storage space per day per computer. Note that the observed disk writes included disk writes to system files as well as user files. The log will be smaller if we record only the disk writes to partitions in the hard disk reserved for user files. Also, by logging only incremental changes to the contents of the data blocks instead of logging copies of entire data blocks whenever disk writes occur, we can further reduce the storage space requirement.

5. Conclusion and Future Work

In this paper, we have presented a methodology to gather digital evidence. We can obtain a consistent snapshot of a file as it existed at a time instant *t* using layers of byteprints of the file's inode and data blocks. These byteprint layers and their corresponding timestamps can be obtained using effective logging mechanisms. Logging modifications to small user-created files is feasible in today's systems. Also, the use of logging techniques avoids making major changes to the existing file system structure. We also presented an application of obtaining such a consistent snapshot, namely, to find whether a given file ever existed on disk.

As future work, we intend to implement a tool that uses the logging mechanism to recover digital evidence using the algorithm given in Section 3.2. Among other things, the tool may be used to recover files that were accidentally deleted by users or to verify if a given file ever existed in the hard disk. The digital evidence thus gathered by this tool may be useful in prosecuting cyber criminals. Checking if a file existed between times t_1 and t_2 can be useful to prove innocence.

[1] Each computer has a 2.4GHz Pentium 4 processor with 80GB of hard disk space, and runs Windows 2000 or Windows XP.

References

[1] M. J. Bach. *The Design of the Unix Operating System*. Prentice Hall, 1986.

[2] E. Casey. *Digital Evidence and Computer Crime: Forensic Science, Computers and the Internet*. Academic Press, London, 2000.

[3] E. Casey. Practical approaches to recovering encrypted digital evidence. *International Journal of Digital Evidence*, 1(3), Fall 2002.

[4] A. Crane. Linux ext2fs undeletion mini-HOWTO. *The Linux Documentation Project*, v1.3, 2 February 1999.

[5] D. Farmer and W. Venema. The coroner's toolkit (TCT). *Doctor Dobb's Journal*, 1999.

[6] A. Frisch. *Essential System Administration, third edition*. O'Reilly and Associates, Inc., 2002.

[7] S. Garfinkel, D. Weise, and S. Strassmann. *The Unix-Haters Handbook*. IDG Books, 1994.

[8] R. Glover. HOW-TO : undelete linux files (ext2fs/debugfs). *comp.os.linux.misc Usenet posting*.

[9] R. Gomez, A. Adly, I. Mayergoyz, and E. Burke. Magnetic force scanning tunneling microscope imaging of overwritten data. *Magnetics, IEEE Transactions on*, 28(5):3141–3143, September 1992.

[10] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of Sixth USENIX Security Symposium*, July 22-25 1996.

[11] K. Hatfield. Unix secure file deletions. *As part of GIAC practical repository*.

[12] N. Majors. Data removal and erasure from hard disk drives. *Tech Articles: ActionFront Data Recovery Labs Inc.*

[13] I. D. Mayergoyz, C. Serpico, C. Krafft, and C. Tse. Magnetic imaging on a spin-stand. *Journal of Applied Physics*, 87(9):6824–6826, May 2000.

[14] S. Powers, J. Peek, T. O'Reilly, and M. Loukides. *UNIX Power Tools*. O'Reilly and Associates, Inc., 2002.

[15] C. H. Sobey. Recovering unrecoverable data: The need for drive-independent data recovery. *Channel Science*, April 2004.

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0-7695-2315-3/05 $ 20.00 **IEEE**

IEEE COMPUTER SOCIETY