available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

# An efficient technique for enhancing forensic capabilities of Ext2 file system

*Mridul Sankar Barik*[a,*], *Gaurav Gupta*[b,1], *Shubhro Sinha*[c], *Alok Mishra*[c], *Chandan Mazumdar*[a]

[a]*Department of CSE, Jadavpur University, Kolkata 700032, India*
[b]*Advisor, IT Advisory, KPMG, India*
[c]*Department of IT, Bengal Engineering and Science University, Shibpur 711103, India*

*Keywords:*
Electronic documents
Modification access and creation date and time stamps (*MAC DTS*)
Authentic date and time stamps (ADTS)
Computer Frauds and Cyber Crimes (CFCC)
Ext2 file system
Loadable Kernel Module (LKM)

ABSTRACT

As electronic documents become more important and valuable in the modern era, attempts are invariably made to take undue-advantage by tampering with them. Tampering with the *m*odification, *a*ccess and *c*reation *d*ate and time stamps (*MAC DTS*) of digital documents pose a great threat and proves to be a major handicap in digital forensic investigation. Authentic date and time stamps (*ADTS*) can provide crucial evidence in linking crime to criminal in cases of Computer Fraud and Cyber Crimes (*CFCC*) through reliable time lining of digital evidence. But the ease with which the *MAC DTS* of stored digital documents can be changed raises some serious questions about the integrity and admissibility of digital evidence, potentially leading to rejection of acquired digital evidence in the court of Law. *MAC DTS* procedures of popular operating systems are inherently flawed and were created only for the sake of convenience and not necessarily keeping in mind the security and digital forensic aspects. This paper explores these issues in the context of the Ext2 file system and also proposes one solution to tackle such issues for the scenario where systems have preinstalled plug-ins in the form of Loadable Kernel Modules, which provide the capability to preserve *ADTS*.

© 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.

## 1. Introduction

Electronic documents of digital era provide the cost effective means to prepare portable, editable, perfectly replicable or mutable documents. The presence of e-documents is evident everywhere right from the accounting processes, to the database handling, and other commonly used documents including emails and web based files. As e-documents attain the importance and significance of immense value in modern era, attempts are invariably made to take undue-advantage by tampering with them.

Tampering frauds comprise firstly, the change in metadata and contents of the file for which existing tools, in most of the cases, provide satisfactory solutions. Secondly, the tampering of *m*odification, *a*ccess and *c*reation *d*ate and time stamps (*MAC DTS*) of digital documents (Boyd et al., 2004; Hosmer, 1998; Weil, 2002; Hosmer, 2002) with ease pose a great threat and proves to be a major hurdle in digital forensic investigation. *MAC DTS* entities reveal the fundamental information regarding when the file was first created and the following modification and access information, which plays an important role in reconstruction of sequence of events in digital

* *Corresponding author.*
E-mail addresses: msbarik@cse.jdvu.ac.in (M.S. Barik), gauravg@kpmg.com (G. Gupta), shubhro22@gmail.com (S. Sinha), alokmishra_besu@yahoo.com (A. Mishra), chandanm@vsnl.com (C. Mazumdar).
[1] The views expressed in this research paper are those of authors only and do not represent the position of KPMG.
1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.007

forensic cases. Thus authentic date and time stamps (*ADTS*) could provide crucial evidence in linking crime to criminal(s) in cases of Computer Frauds and Cyber Crimes (*CFCC*) through reliable time lining of digital evidence. Most countries recognize e-document as legitimate legal documents. Electronic documents have been used as substantial sources of evidence in various cases of national and international importance. The ease with which *MAC DTS* of stored digital documents could be changed raises some serious questions about the integrity and admissibility of digital evidence (e.g. how a file can be modified before it was created?) potentially leading to rejection of acquired digital evidence in the court of Law.

Digital forensic aspects of e-documents (e-document forensics) require urgent attention due to its impact in solving most of the cases of *CFCC*, especially issues related to authentic date and time stamps.

All currently available date and time stamp procedures simply acquire available date and time stamps of the system at that particular instance, which are easy to change. Hence existing *MAC DTS* procedures are inherently flawed. These procedures were created only for the sake of convenience and not keeping in mind security and digital forensic aspects.

This paper explores the issues related to the date and time stamps of stored digital documents in the Ext2 file system. We propose a solution to tackle such issues in the Ext2 file system, for the scenario where we have pre-installed plug-ins, which provide the capability to preserve *ADTS*.

## 2. Related work

The ease of availability of methods and tools to change *MAC DTS* allows criminals to exploit this vulnerability to their maximum advantage. For example, criminals may tamper with *MAC DTS* as per their wish to gain unlawful advantage, to implicate other innocent persons, to gain unlawful financial access and most importantly, to put a question mark on admissibility and integrity of digital evidence, using commonly used APIs and system utilities. Various methods of time stamping, viz. using network time protocol (*NTP*), checksum like CRC 16 and CRC 32, one way hash algorithms like SHA-1, MD 2, MD 4, MD 5 (Russell, 2001), digital signatures, Secure Time Stamp Device (Forensic Examination, 2004) have been used, where secure time is acquired from secure server and is digitally signed with the respective files. But these methods do not confirm the absolute authenticity of the documents. Then they cannot answer whether the *MAC DTS* associated with the file is real? Have they been tampered with or not? They can provide the information that the file(s) in question has not been changed since they were last digitally signed but *MAC DTS* resides in FAT, MFT and inode's and hence it does not guarantee the authenticity of the documents under consideration. Also the available breed of digital forensic analysis tools like EnCase, ilook cannot guarantee the tamper-proof-ness of available date and time stamp of the files.

Traditional journaling file systems were designed keeping in mind the ease of recovery after a system crash and not exactly to help in forensic investigation. They usually keep a log of only the recent transactions committed or to be committed on the data and the metadata of the file system. Once the file system transaction has been committed the journal entry of the journaling file systems becomes useless and on being sufficiently old is removed from the journal. For example the Ext3 (Farmer and Venema, 2004; Red Hat) file system can maintain journals of both file system data and metadata (which includes MAC times) and is of limited size. This leaves forensic investigators with very little MAC information in case it is configured to journal both data and metadata. Also the journal is stored as a regular file (a hidden file in the root directory) making it vulnerable to tampering. Other journaling file systems like Reiser, XFS, JFS (IBM), etc. generates journals of activities affecting metadata only. The Linux Audit Subsystem for kernel 2.6 generates audit records which are stored in regular files and contain only last MAC information among other things.

## 3. Solution approaches

As file systems overwrite the changes in *MAC DTS*, we cannot get back the original *MAC* details, as the retrieval of overwritten data is very difficult and expensive. Hence what we want is the capability in file system itself to assist in preserving original *MAC DTS* to help in digital forensic investigation. This approach requires either development of a file system with the capability of preserving original *MAC DTS* with authenticity from scratch, or deployment of a plug-in which will help in authentically preserving *MAC DTS*.

As criminals use existing operating systems which as of now do not have these capabilities, researchers are exploring solution from two different perspectives. The first solution deals with the cases where we do not have any pre-installed software and we try to retrieve discrepancies relying on basic principle of forensic science. Second solution deals with the cases, where we have either a pre-installed plug-in, or a kernel with enhancements to support *ADTS*. In this paper we present a solution in the second category. Our solution is based on Ext2 file system and uses pre-installed plug-in, in form of Loadable Kernel Modules.

## 4. Background

The *Second Extended Filesystem* (*Ext2*) (Bovet and Cesati; Card et al., 1994) introduced in 1994 has gained reputation as the most widely used Linux file system due to its efficiency and robustness. For this reason we have chosen Ext2 file system for our case study. The following paragraphs briefly describe the layout of an Ext2 file system and the concept of VFS.

As shown in Fig. 1, the first block in an Ext2 partition is reserved for the partition boot sector. The remaining blocks of the partition are split into number of *block groups*. All the block groups in the file system are of same size and are stored sequentially, so the kernel can derive the starting block number of a block group in a disk simply from its integer index. The concept of block groups reduces file fragmentation, since the kernel tries to keep the data blocks belonging to a file in the same block group if possible. Each block group consists of the following:

- A copy of the file system's superblock.
- A copy of the block group descriptors.

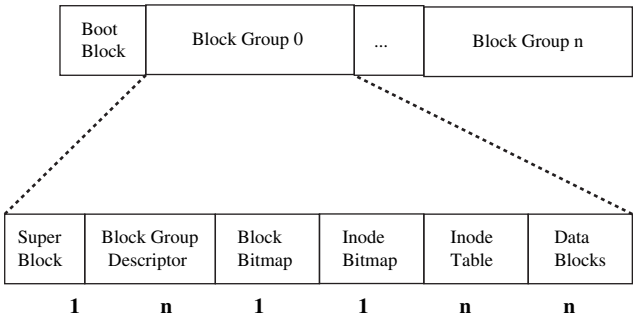



**Fig. 1 – Layout of Ext2 file system.**

- A data block bitmap.
- An inode bitmap.
- A group of inodes.
- Data blocks.

The number of block groups in an Ext2 file system depends both on the partition size and on the block size. However, there is a constraint that the block bitmap, which is used to identify the blocks that are used and free inside a group, must be stored in a single block.

One of Linux's keys to success is its ability to coexist comfortably with other systems. It allows users to transparently mount disks or partitions that host file formats used by Windows or other UNIX systems. To support this feature Linux uses a concept called the Virtual File System (VFS) which is a kernel software layer that handles all system calls related to a standard UNIX file system. Its main strength is providing a common interface to several kinds of file systems.

The key idea behind VFS consists of introducing a *common file model* capable of representing all supported file systems. For instance, in this model, each directory is regarded as a file, which contains a list of files and other directories. However, several non-Unix disk-based file systems use a File Allocation Table (FAT), which stores the position of each file in the directory tree. These file systems, do not treat directories as files. To satisfy VFS's common file model, the Linux implementations of such FAT-based file systems need to construct on the fly, the files corresponding to the directories and such files exist only as objects in kernel memory.

Actually, the Linux kernel does not hardcode a particular function to handle an operation such as read() or ioctl(). Instead, it uses a pointer for each operation and the pointer is made to point to the proper function for the particular file system being accessed.

For example, if we want to read a file in an MS-DOS file system, the kernel converts the *read()* system call to a call of MS-DOS implementation of *read()*. This conversion is done by a hierarchy of data structures associated with Virtual File System.

The common file model consists of the following data structure:

- *Superblock object*: stores information about a mounted file system.
- *Inode object*: stores general information about specific file.
- *File object*: stores information about the interaction between an open file and a process.
- *Dentry object*: stores information about the linking of a directory entry with the corresponding file.

Fig. 2 demonstrates the relation between a process and various VFS objects.

## 5. Design and implementation

To implement MAC trail logging mechanism in the Ext2 file system we have kept in mind the following four objectives:

1. We should be able to store the MAC trail however long the trail may be.
2. MAC trails should be kept in such a manner that normal users do not even get an idea about existence of such a trail logging mechanism in their file system.
3. The trail should be easily accessible to the administrator.
4. The MAC trail logging mechanism should not degrade the system performance considerably.

In our design, we have made an assumption that, the administrator will be interested in keeping MAC trail of critical files only. Our design makes a provision whereby the administrator can opt for those file that are to be monitored. We have introduced a new system call for this purpose.

When a file is being monitored, each invocation of *sys_open* system call over this file would generate a record in MAC trail of that file, describing this access. To understand this fact let us first look at how a file is accessed (Bovet and Cesati; Bach, 1988). Every regular file is accessed via the *sys_open* system call. When a user process wants to open a file for reading or writing it makes a call to the *sys_open* system call either explicitly or implicitly. By explicit calling we mean that it directly calls *sys_open*, where as by implicit open we mean that the system call is invoked by some other program on behalf of the user. The *sys_open* or more commonly known by its 'C' wrapper function *open* finds an unused file descriptor in the processes file descriptor table. The lowest numbered file descriptor is returned to the process. Fig. 3 shows actions performed during invocation of sys_open system call.

In our implementation, we have modified the *sys_open* system call with the help of Loadable Kernel Modules (*LKMs*). This

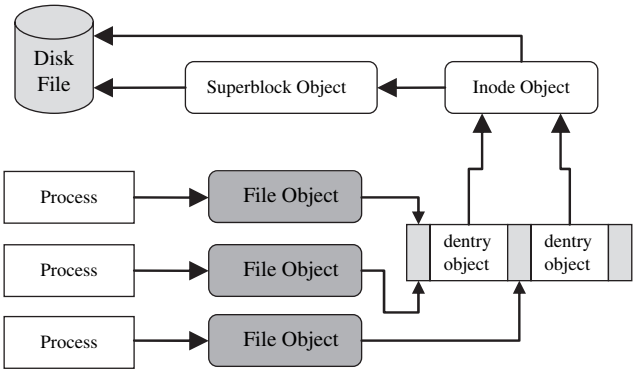
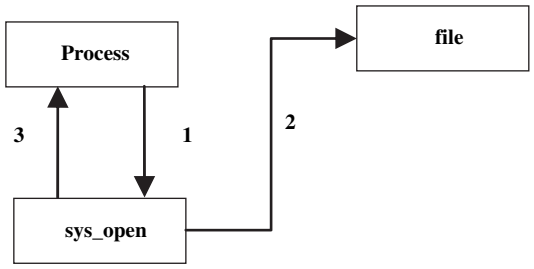



**Fig. 2 – Interaction between processes and VFS objects.**




1. User process calls the sys_open system call
2. sys_open performs the following tasks:
   a. Locates the inode and does various checks.
   b. If the UID of the process is allowed to access the file, create a file table entry in the file descriptor table of the process and record the offset
3. Return this offset as the file descriptor to the process

**Fig. 3 – Actions of *sys_open*.**

also provides flexibility in our implementation as this logging mechanism can be inserted and removed at will of the administrator.

The information regarding where in the file system the MAC trail log of individual files start, is kept in a data structure, known as MAC directory structure. This directory structure links unique identifier of individual files to the starting block number of that file's MAC trail. An obvious choice for the unique identifier for Ext2 file system is the inode number, which is the index of the inode structures in the inode table.

The MAC trails are also required to be hidden from the users. Unlike VFS image of file system directory structure, the MAC directory structure does not have any VFS image. Rather some unused VFS inode fields are used to keep file specific MAC trail information, i.e. the address of the MAC trail block where the current MAC record of that file is to be written. Also the MAC trails are written onto the disk in random blocks to reduce the probability of someone finding out the existence of the trails. These random blocks are linked to each other in form of a doubly linked list.

### 5.1. *Data structures used*

In this section we describe the data structures of the MAC directory blocks and MAC trail blocks.

The structure of an MAC directory block is shown in Fig. 4. Each MAC directory block starts with an 8 bytes identifier designating the block as an MAC directory block. This helps in reconstructing the MAC directory, in case the linked list is corrupted. The following two 4 bytes long fields contain pointers to the previous and next MAC directory blocks. This is followed by number of MAC directory entries each of size 8 bytes. The exact number of entries in an MAC directory block obviously is dependent on the size of the block itself.

An MAC directory block entry is a tuple <inode number, MAC trail starting block>. The structure of an MAC directory block entry is shown in Fig. 5.

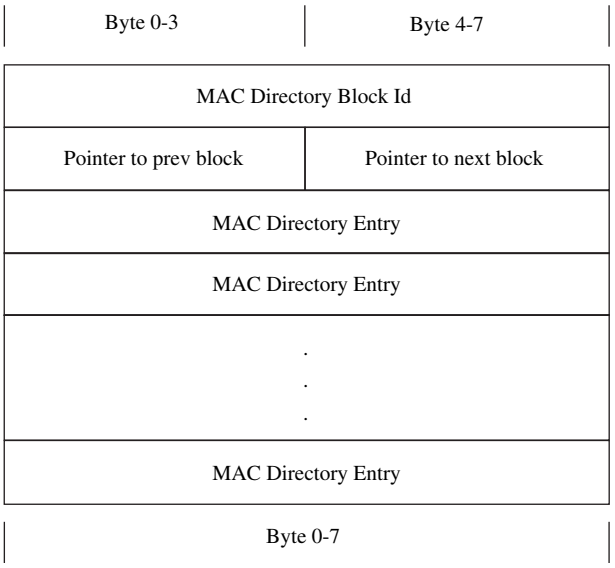



**Fig. 4 – Structure of an MAC directory block.**

The structure of each MAC trail block as shown in Fig. 6 is similar to the MAC directory block except that each MAC trail entry is 12 bytes long.

The structure of each MAC trail entry is shown in Fig. 7.

Now that we have given the details of the data structures we can move on to the operation on these data structures.

### 5.2. *Operations on data structure*

In order to reduce search time we have kept MAC directory entries of all monitored files whose first data block lies in a specific block group, in a single MAC directory block linked list. In fact, the block group descriptor of a block group keeps a pointer to the first block of the MAC directory containing MAC directory entries for all monitored files whose first data block lies in that specific block group. We have used the field *bg_reserved[0]* in the block group descriptor for this purpose.

We have introduced a system call *init_activites* to bring all MAC trail metadata in to main memory. During system boot time a program is run which calls this system call for all files that are to be monitored. The activities performed by the system call are:

1. Locate the VFS inode for the file passed as parameter.
2. From VFS inode locate the first data block of the file.
3. Determine the block group no of the first data block of the file and from super block get the corresponding block group descriptor.
4. Get the MAC directory block address for the file from the value contained in *bg_reserved[0]* field in the block group descriptor obtained in step 3.

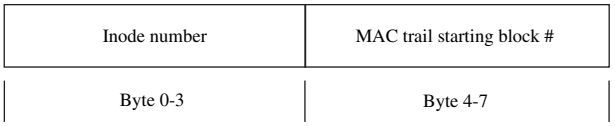



**Fig. 5 – Structure of MAC directory block entry.**

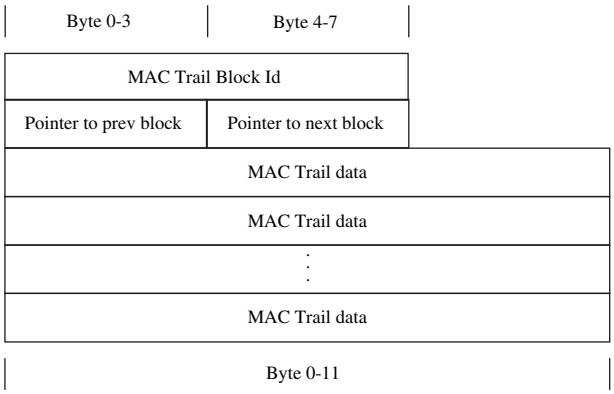



**Fig. 6 – Structure of an MAC trail block.**

5. If an entry corresponding to the file inode number is found in the MAC directory, fetch that entry and determine the MAC trail block number where the next MAC record will be written. Put that block number into the *i_generation* field of the VFS inode. Also set the value of *i_sock* to nonzero.
6. Increment the VFS inode usage counter so that the inode stays in the memory the entire duration the system is up.

Note that the fields *i_generation* and *i_sock* are not used in case of regular files. So, we have used it comfortably in our MAC trail logging system. In fact, *i_generation* is used in case of NFS and *i_sock* is nonzero in case of sockets.

We have modified the *sys_open* system call such that before returning the file descriptor to the user process it performs the activities briefed in Fig. 8.

Following describes activities (starting from activity no. 3) of modified *sys_open* system call in detail as shown in Fig. 8.

*Locate VFS inode of the file*: the open system call itself locates the VFS inode to check for file permissions, etc. We have used this already located inode.

*Check whether the file is monitored or not*: we have used the *i_sock* field in the VFS inode for this purpose of designating a file as being monitored. If *i_sock* field has nonzero value it implies that the file is being monitored.

*Locate the MAC trail writing block*: if a file is marked as monitored, we fetch the MAC trail block where the current record will be written. We have used the *i_generation* field of the VFS inode for this purpose.

*Generate an MAC record and write into the MAC block*: the inode structure contains the last MAC times. The entries are formatted in a proper way and written into the MAC trail block fetched in the previous step. If the block does not have any room then we allocate a random block, and write MAC record into that block and update the *next_block* and *prev_block* fields of the old and new block, respectively. We also update the value of *i_generation* field to the new block number.

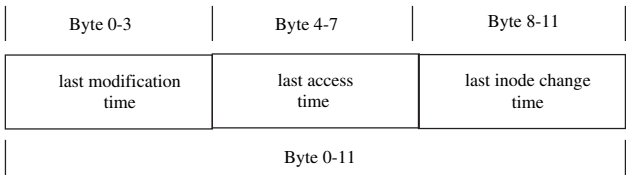
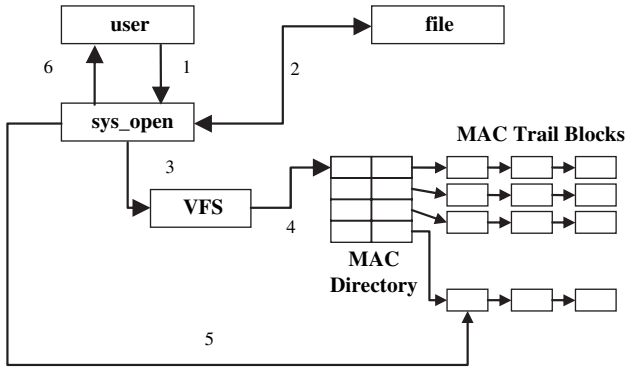



**Fig. 7 – Structure of an MAC block entry.**




1. The userprocess calls *sys_open*
2. *sys_open* system call performs its normal operations i.e. checking file permissions etc. and if the access to the file is granted the file descriptor is recorded.
3. Locates the VFS inode of the file and check whether the file is monitored or not.
4. Obtains the pointer to the MAC trail block for writing the current record from the VFS inode of the file.
5. Generates a MAC trail record for the file usage and writes the record to the MAC trail block.
6. Returns the recorded file descriptor to the user

**Fig. 8 – Actions of modified *sys_open*.**

### 5.3. *Algorithms*

The function *locate_mac_dir_list* returns a pointer to the head of the MAC directory list of a monitored file.

```
function locate_mac_dir_list
input: file_path
output: Pointer to the MAC directory list for that file
{
      inode = get VFS inode of the file;
      fdb = inode->first data block of the file;
      gdp = get_group_desc(fdb);
      return (gdp->bg_reserved[0]);
}
```

The function *get_group_desc* returns the block group descriptor for a given block number.

```
function get_group_desc
input: block number
output: Pointer to block group descriptor
{
      bg_no = block_no/blocks_per_block_group;
      search the superblock, for the block group
            descriptor of bg_no;
      return group_descriptor.
}
```

The function *get_free_block* finds a random block and returns a pointer to it if it is free.

```
function get_free_block
input: file system super_block
output: available free block
{
    do
    {
        r_no=generate a random number;
    }while(the block r_no is not free);
    return (r_no);
}
```

The algorithm of *sys_open* system call which we have modified for the MAC trail logging mechanism is given below.

```
System call modified_sys_open
input: file_path_name, flags, mode,
output: file_descriptor
{

    perform normal sys_open activities to get an
        unused file_descriptor;

    inode=locate the VFS inode of the file;
    check whether the file is monitored or not by
        looking at specific fields in inode;
    if(monitored)
    {
        generate the MAC trail record;
        locate the last MAC trail block from the
            inode;
        if(last MAC trail block not full)
            write MAC trail record;
        else
        {
            get_free_block (inode->superblock);
            link the new block with the last block of
                the list;
            update inode to point to the new block;
            write the MAC trail record to the new
                block;
        }
    }

    return file_descriptor;
}
```

As stated earlier, to speed up MAC trail writing process, we have kept a pointer to the MAC trail block where the current record of the monitored file has to be written, in the VFS inode of that file itself. The *init_activites* system call caches the pointer to the current MAC trail blocks of all files in their VFS inodes.

```
System call init_activities
input: file_path
output: none
{
    inode = locate the VFS inode of the file;
    inode = VFS inode;
    ino = inode->inode_number;
    dir_blocklist = locate_dir_block_list(file_path);
    search entry of ino;
    mark inode as monitored;
    cache MAC trail block into VFS inode;
    return;
}
```

To submit a file for monitoring we have created a system call *submit_file* which takes the file path as an argument. The algorithm is as follows:

```
System call submit_file
input: file_path
output: none
{
    inode=locate the VFS inode of the file;
    ino = inode->inode_no;
    gdp = get_group_desc (inode->first data block of the
                          file);
    if(gdp->bg_reserved[0] == 0)
    /*No MAC directory block exist for the file or for
    that matter all files whose first data block lie in the
    same block group*/
    {
        dir_blk = get_free_block (inode->super block);
        gdp->bg_reserved[0] = dir_blk;
    }
    else
        dir_blk=gdp->bg_reserved[0];

    if(no MAC directory entry corresponding to ino is
    found)
    {
        /*allocate a MAC trail block for the file*/
        trail_blk = get_free_block
                    (inode->superblock);
    }
    else
        return;

    insert <ino, trail_blk> to dir_block;
    mark file as monitored;
    return;
}
```

The proposed mechanism has been implemented and tested in both Ext2 and Ext3 file systems with Linux kernel version 2.4. But with Linux kernel version 2.6 this mechanism does not work as it does not support modification of the *sys_call_table* kernel data structure.

## 6. Conclusion

The very purpose of Law enforcement and other investigative agencies is to link crime to the criminal(s) and *ADTS* of an electronic document provides the most crucial and fundamental evidence and thus plays an important role in the

reconstruction of sequence of events and in turn, in digital forensic investigation. The inconsistency in date and time stamp raises some serious questions on the integrity and admissibility of potential legal digital evidence (*PLDE* ). As it is not possible to immediately change the file system in use, this paper presents a solution for obtaining authentic date and time stamps of digital documents in question. The implemented solution is based on Ext2 file system and uses preinstalled plug-in in the form of Loadable Kernel Modules (LKMs). This is advantageous in the sense that it provides flexibility as this logging mechanism can be inserted and removed as per will of the administrator. Also it doesn't require any change in existing installations.

One drawback with our solution is that invocation of *utime* system call does not generate any MAC entry. But of course the *utime* system call can be modified to generate an MAC entry. Also the LKM hiding techniques can be used to further enhance the security of the proposed solution itself.

As future work we are planning to design a new file system based on the architecture of Ext2 file system, which will incorporate the forensic capabilities described in this paper.

## Acknowledgement

Authors are thankful to the anonymous reviewers who provided valuable suggestions on improvement of this paper.

## REFERENCES

Boyd Chris, Forster Pete. Time and date issues in forensic computing: a case study. National Technical Assistance Centre, UK. Digit Investig 2004;1:18–23 [Elsevier].

Bovet Daniel P, Cesati Marco. Understanding the Linux kernel. 2nd ed. O'Reilly & Associates.

Bach Maurice J. The design of the UNIX operating system. Prentice Hall of India; 1988.

Card Rémy, Ts'o Theodore, Tweedie Stephen. Design and implementation of the second extended filesystem. In: Proceedings of the first Dutch international symposium on Linux, 1994. ISBN 90-367-0385-9. Available from: <http://web.mit.edu/tytso/www/linux/ext2intro.html>.

Encase® Forensic. Guidance software, <http://www.guidancesoftware.com>.

Farmer D, Venema W. Forensic discovery. Addison Wesley; 2004.

Forensic examination of digital evidence: guide for law enforcement. U.S. Department of Justice, Office of Justice Programs, National Institute of Justice; April 2004.

Hosmer Chet. Time lining computer evidence. WetStone Technologies. In: Information Technology Conference, IEEE; 1998. p. 109–12.

Hosmer Chet. Proving the integrity of digital evidence with time. President & CEO, WetStone Technologies, Inc. Int J Digit Evid Spring 2002;1(1)

Ilook Investigator©. Computer forensics software, <http://www.ilook-forensics.org>.

IBM's journaled filesystem, <http://www.linuxjournal.com/article/6268>.

Linux audit-subsystem design documentation for kernel 2.6 version 0.1. Available from: <http://www.uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf>.

Russell Chris. Analysis of a secure time stamp device. In: GSEC practical assignment version 1.2f, for GIAC certification in security essentials October 17, 2001. SANS Institute; 2001.

Red Hat's new journaling file system: Ext3, <http://www.redhat.com/support/wpapers/redhat/ext3/>.

Reiser file system, <http://www.namesys.com/>.

Weil Michael C. Dynamic time & date stamp analysis. Computer Forensic Examiner, Department of Defense Computer Forensics. Int J Digit Evid Summer 2002;1(2).

XFS: A high-performance journaling filesystem, <http://oss.sgi.com/projects/xfs/>.

**Mridul Sankar Barik** is presently a Lecturer in the Department of Computer Science and Engineering, Jadavpur University, India. His research interests include Digital Forensics, Information Security and Distributed Systems.

**Gaurav Gupta** is presently pursuing his doctorate on the topic "Studies on Digital Forensics to Detect Computer Frauds and Cyber Crimes" in the Department of Computer Science, Jadavpur University, India. He is also working as an advisor with IT Advisory of KPMG, India. His research interests include Information Security, Digital Forensics and its applications.

**Shubhro Sinha** is a final year student of Bachelor of Engineering, Department of Information Technology, Bengal Engineering And Science University, Shibpur, India. His research interests include Digital Forensics and Operating System Principles.

**Alok Mishra** is a final year student of Bachelor of Engineering, Department of Information Technology, Bengal Engineering And Science University, Shibpur, India. His research interests include Digital Forensics and Operating System Principles.

**Chandan Mazumdar** is presently a Professor in the Department of Computer Science & Engineering, and is the Coordinator of the Center for Distributed Computing, Jadavpur University, India. His research interests include Information Security, Digital Forensics, and Dependable Computing. He has copy-righted two software: one on Forensic Identification of Fire Arms and another on Enterprise Information Security Management. He was Program Co-chair of the First International Conference on Information Systems Security (ICISS), 2005.