available at www.sciencedirect.com

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

Digital Investigation

Forensic discovery auditing of digital evidence containers

Golden G. Richard III[*], *Vassil Roussev*, *Lodovico Marziale*

Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

ARTICLE INFO

Article history:
Received 13 February 2007
Accepted 2 April 2007

Keywords:
Digital forensics
Operating systems internals
Filesystems
Digital evidence containers
Auditing

ABSTRACT

Current digital forensics methods capture, preserve, and analyze digital evidence in general-purpose electronic containers (typically, plain files) with no dedicated support to help establish that the evidence has been properly handled. Auditing of a digital investigation, from identification and seizure of evidence through duplication and investigation is, essentially, ad hoc, recorded in separate log files or in an investigator's case notebook. Auditing performed in this fashion is bound to be incomplete, because different tools provide widely disparate amounts of auditing information – including none at all – and there is ample room for human error. The latter is a particularly pressing concern given the fast growth of the size of forensic targets. Recently, there has been a serious community effort to develop an open standard for specialized digital evidence containers (DECs). A DEC differs from a general purpose container in that, in addition to the actual evidence, it bundles arbitrary metadata associated with it, such as logs and notes, and provides the basic means to detect evidence-tampering through digital signatures. Current approaches consist of defining a container format and providing a specialized library that can be used to manipulate it. While a big step in the right direction, this approach has some non-trivial shortcomings – it requires the retooling of existing forensic software and, thereby, limits the number of tools available to the investigator. More importantly, however, it does not provide a complete solution since it only records snapshots of the state of the DEC without being able to provide a trusted log of all data operations *actually* performed on the evidence. Without a trusted log the question of whether a tool worked exactly as advertised cannot be answered with certainty, which opens the door to challenges (both legitimate and frivolous) of the results.

In this paper, we propose a complementary mechanism, called the *Forensic Discovery Auditing Module* (*FDAM*), aimed at closing this loophole in the discovery process. FDAM can be thought of as a 'clean-room' environment for the manipulation of digital evidence, where evidence from containers is placed for controlled manipulation. It functions as an operating system component, which monitors and logs all access to the evidence and enforces policy restrictions. This allows the immediate, safe, and verifiable use of *any* tool deemed necessary by the examiner. In addition, the module can provide transparent support for multiple DEC formats, thereby greatly simplifying the adoption of open standards.

1. Introduction

Information discovered during an investigative inquiry becomes usable evidence only when it passes the rigorous test of admissability. One of the crucial aspects of arguing admissability is demonstrating that the information has been obtained and handled properly and, therefore, the obtained results can be trusted. In traditional forensics, where physical

[*] *Corresponding author*. Tel.: +1 504 957 5814.
E-mail addresses: golden@cs.uno.edu (G.G. Richard III), vassil@cs.uno.edu (V. Roussev), lmarzial@cs.uno.edu (L. Marziale).

1742-2876/$ – see front matter © 2007 Published by Elsevier Ltd.
doi:10.1016/j.diin.2007.04.002

artifacts are examined and interpreted, sealed evidence bags are a simple but indispensible tool in making that argument. In addition to tamper-evident design, such bags often provide writing space for identifying information, such as the name of the investigating officer, case identifiers, the suspect's name, a description of the item, and the date and time when the bag was sealed. Continuity sections on the bag allow tracking the movement of the bag, noting the chain of custodians who have undertaken the bag's care.

Digital forensic examiners have generally followed a similar approach, with one notable exception – so far they lack the equivalent of a tamper-evident bag for digital information. Currently, the state of the art digital forensic tools operate over standard operating systems' components and use plain files as containers for holding artifacts. One of the problems is that general-purpose tools used during the forensic process might potentially alter the artifact. Although this can be prevented by a write blocker, the issue of documenting which operations the tool performed remains unanswered. For example, if a tool was used to search for malware, was the whole disk searched, or just selected 'typical' places? To deal (in part) with this problem, best practices dictate that all work be performed on a copy of the evidence and all operations performed be recorded in an audit log. For the purposes of this article, we refer to the process of keeping a log of all operations on the evidence as *auditing*.

Under current methodology, which places the burden entirely on the human investigator, auditing is bound to be incomplete, because different tools provide varying levels of auditing information (in a variety of formats), much of which must be recorded manually. Over the course of an investigation, a piece of digital evidence may be touched by many different tools, some of which generate no audit trail of their actions (e.g., *dd* or the command line tools of the Sleuthkit) and some that generate their own audit logs (e.g., FTK). At the end, an investigator is left to piece together these bits of audit trail to create a comprehensive view of what occurred during the investigation. Another potential issue with application-generated logs is that the application is charged with policing itself, which makes the possibility of both the application and its log being faulty much more likely than if they were developed separately. Finally, as a practical concern, failure to manually record a bit of auditing information, such as the MD5 hash generated by *md5sum* for a large disk image, could potentially result in a huge amount of lost time if the operation must be repeated. If generation of auditing details such as these is automated, so much the better.

Anecdotal evidence suggests that many examiners try to circumvent the above problem by using only the results produced by commercial tools of vendors prepared to vigorously defend their product in court (at own expense). There are at least two major problems with this approach. The first one is that investigators are limited to the capabilities of the chosen tools. In general, there is hardly anything special about the functions found in most forensic tool suites – almost all of these are based on functions existent in general-purpose tools (e.g., data transfer, text searches, file type identification, etc.) and, over time, forensic tools simply accumulate more of them in a convenient package. Nonetheless, there will always be a need to use other tools to discover and interpret evidence.

Another issue is that, just like any other piece of software, forensic tools invariably contain implementation errors (bugs) or are based on unsound assumptions. As a simple example, earlier versions of many tools could be fooled into believing that a text file was a Microsoft Windows executable by starting the file with the string 'MZ'. It is not always the vendor's fault – artifacts created in proprietary formats (e.g., many formats in Microsoft Windows) can change with no warning from version to version. While discovered bugs and shortcomings tend to be fixed, the continuous introduction of new tool features brings new potential problems. For example, recent trends towards multi-threading bring into the picture a whole new class of *potential* implementation problems related to synchronization that do not exist in a single-threaded implementation.

In short, both investigators and the tools they use are prone to errors and this can lead to challenges (both legitimate and frivolous) of the results. Since the possibility of error will never go away and is inherently difficult to quantify, the only practical way to genuinely improve the trustworthiness of the process is to have an independent auditing facility that is: (a) automated, to guard against human lapses and (b) tool-independent, to independently confirm/challenge the results of any tool used, including ones not specifically labeled 'forensic'.

We should emphasize that such an independent auditing facility is not charged with duplicating the forensic functions of any tool but is an impartial observer of all basic data operations *actually* performed on the evidence. The essential result is a consistent, trusted, and tamper-evident audit trail that can be examined after the fact to confirm/debunk challenges.

Recently, Turner (2005) coined the rather descriptive term Digital Evidence Bag (DEB) to describe his proposed approach to dealing with the above problems. More precisely, his work can be described as an effort to create a specialized container – an open, common file format – for storing digital evidence. DEBs bundle digital evidence, associated metadata, and audit logs into a single structure, providing an audit trail of operations performed on the digital evidence in the bag as well as integrity checks. In the following section, we describe DEBs and other *digital evidence containers* (or *DECs*) in more detail.

All current DEC specifications address the auditing problem to some degree. However, they all rely on an individual tool's 'voluntary' participation, i.e., forensic applications need to use a specialized API, which effectively replaces the filesystem API. Obviously, reengineering all existing forensic applications for that specific purpose is a tall order and there are no compelling incentives at the moment. Therefore, our work is targeted at the development of operating system-level mechanisms to support/enforce the use of DECs. As we will demonstrate, this approach can achieve both automation and tool-independence without additional development efforts for existing tools. Overall, our approach can also provide stronger auditing guarantees. We view this as a complementary mechanism to development and standardization of DEC formats, which aids in evidence-handling procedures and provides transparent support for a variety of digital evidence container formats.

2. Related work

In this section, we provide a quick overview of some representative work on DECs without attempting to provide a full survey. We also briefly review some classical work on secure audit logs.

2.1. *Digital Evidence Bags*

The basic structure of Digital Evidence Bags (DEBs) is outlined by Turner (2005) and further developed by Turner (2006). As the example in Fig. 1 shows, a DEB consists of a hierarchical collection of objects (files) that contain digital evidence, along with relevant meta-information accumulated during the investigative process, including audit information.

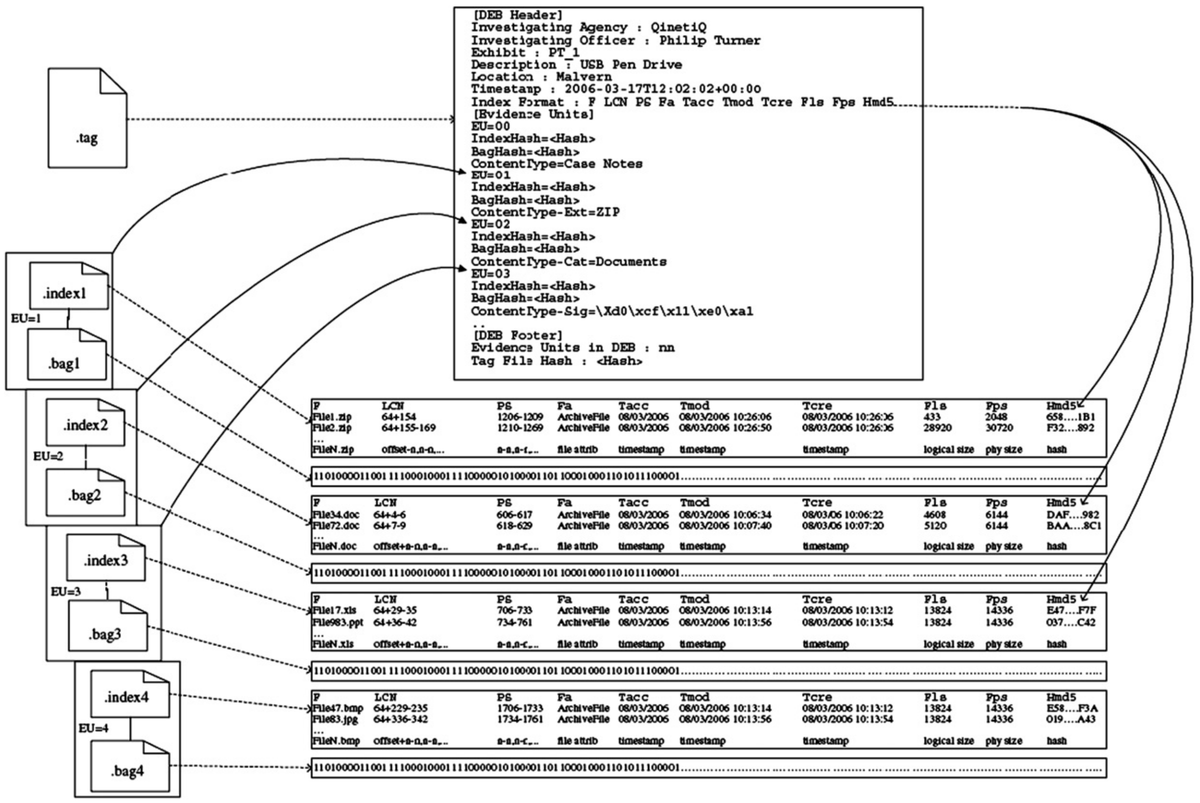
At the root of the hierarchy is the *tag* area, which consists of a set of name/value pairs containing metadata associated with the DEB. This includes a unique identifier for the DEB, the creation date and time, the name and organization of the creator of the DEB, and a list of Evidence Units (EUs) stored in the DEB. With the exception of Evidence Unit 0, which is reserved for case notes and case-associated metadata at the time of DEB creation, the content of each unit can vary, depending on need. EU0 contains information about the target (e.g., type, serial number, and manufacturer) and the imaging tool used to create the DEB (name, version, integrity hash, configuration file, run-time options, etc.) Additional free-form text, images, etc. can be included to document the acquisition process.

Each EU provides a name for a distinct *blob* (piece of binary data) of digital evidence stored in the bag and the blob's associated index file. The index file describes the blob's content in detail, such as file names, hashes, timestamps, and physical layout information. A unit could also be comprised entirely of meta-data, such as a list of file hashes. Finally, an audit log (called a *Tag Continuity Block*) tracks the operations performed against a DEB, including the date, time, affected blocks, and application signature of each operation as well as periodic hashes of the DEB contents.

The DEB design strives for simplicity, with all metadata recorded as plain text and all data recorded in raw binary format. Efficient access and representation are not a major consideration at this time and neither is secure auditing, although we see no technical impediments to adding these later.

2.2. *AFF and Gfzip*

AFF (Garfinkel, 2006) is an ambitious open-source project with a major emphasis on efficiency, both in terms of storage requirements and access time. Furthermore, it has built-in provisions for strong authentication based on an X.509(v)3 certificates. AFF defines disk-representation and data-storage layers, where the former defines segment names used for storing all data associated with an image (data and metadata) and the latter defines how the image is stored (in binary or XML). Generally, an AFF file consists of a header and a number of named metadata segments, followed by the data segments



```
[DEB Header]
Investigating Agency : QinetiQ
Investigating Officer : Philip Turner
Exhibit : PT_1
Description : USB Pen Drive
Location : Malvern
Timestamp : 2006-03-17T12:02:02+00:00
Index Format : F LCN PS Fa Tacc Tmod Tcre Fls Fps Hmd5
[Evidence Units]
EU=00
IndexHash=<Hash>
BagHash=<Hash>
ContentType=Case Notes
EU=01
IndexHash=<Hash>
BagHash=<Hash>
ContentType-Ext=ZIP
EU=02
IndexHash=<Hash>
BagHash=<Hash>
ContentType-Cat=Documents
EU=03
IndexHash=<Hash>
BagHash=<Hash>
ContentType-Sig=\Xd0\xcf\x11\xe0\xa1
..
[DEB Footer]
Evidence Units in DEB : nn
Tag File Hash : <Hash>
```

```
F           LCN              PS         Fa          Tacc        Tmod                 Tcre                 Fls           Fps        Hmd5
File1.zip   64+154           1206-1209  ArchiveFile 08/03/2006  08/03/2006 10:26:06  08/03/2006 10:26:06  433           2048       658….1B1
File2.zip   64+155-169       1210-1269  ArchiveFile 08/03/2006  08/03/2006 10:26:50  08/03/2006 10:26:06  28920         30720      F32….892
...
FileN.zip   offset-n,n-n,... n-n,n-r,.. file attrib timestamp   timestamp            timestamp            logical size  phy size   hash
```

```
1101000011001 1110001000111100000101000011101 100010001101011100001..............................
```

```
F           LCN              PS         Fa          Tacc        Tmod                 Tcre                 Fls           Fps        Hmd5
File34.doc  64+4-6           606-617    ArchiveFile 08/03/2006  08/03/2006 10:06:34  08/03/06 10:06:22    4608          6144       DAF….982
File72.doc  64+7-9           618-629    ArchiveFile 08/03/2006  08/03/2006 10:07:40  08/03/06 10:07:20    5120          6144       BAA….8C1
...
FileN.doc   offset+n-n,n-n,.. n-n,n-c,.. file attrib timestamp  timestamp            timestamp            logical size  phy size   hash
```

```
1101000011001 1110001000111100000101000011101 100010001101011100001..............................
```

```
F           LCN              PS         Fa          Tacc        Tmod                 Tcre                 Fls           Fps        Hmd5
File17.xls  64+29-35         706-733    ArchiveFile 08/03/2006  08/03/2006 10:13:14  08/03/2006 10:13:12  13824         14336      E47….F7F
File983.ppt 64+36-42         734-761    ArchiveFile 08/03/2006  08/03/2006 10:13:56  08/03/2006 10:13:54  13824         14336      037….C42
...
FileN.xls   offset+n-n,n-n,.. n-n,n-c,.. file attrib timestamp  timestamp            timestamp            logical size  phy size   hash
```

```
1101000011001 1110001000111100000101000011101 100010001101011100001..............................
```

```
F           LCN              PS         Fa          Tacc        Tmod                 Tcre                 Fls           Fps        Hmd5
File47.bmp  64+229-235       1706-1733  ArchiveFile 08/03/2006  08/03/2006 10:13:14  08/03/2006 10:13:12  13824         14336      E58….F3A
File83.jpg  64+336-342       1734-1761  ArchiveFile 08/03/2006  08/03/2006 10:13:56  08/03/2006 10:13:54  13824         14336      019….A43
...
FileN.bmp   offset+n-n,n-n,.. n-n,n-c,.. file attrib timestamp  timestamp            timestamp            logical size  phy size   hash
```

```
1101000011001 1110001000111100000101000011101 100010001101011100001..............................
```

Fig. 1 – An example of Digital Evidence Bag (DEB) (Turner, 2006).

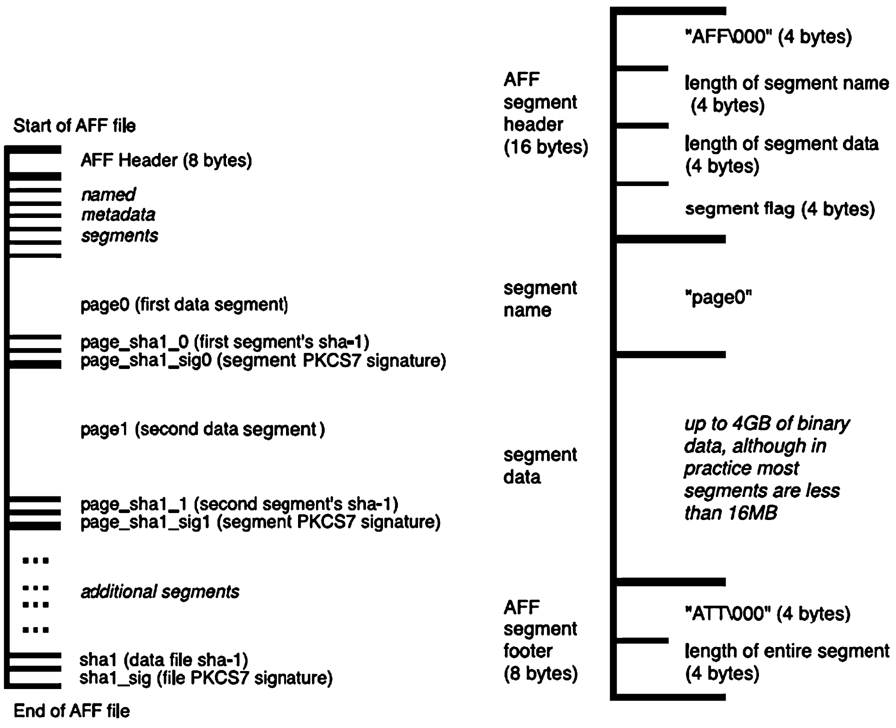

Start of AFF file
AFF Header (8 bytes)
named metadata segments
page0 (first data segment)
page_sha1_0 (first segment's sha-1)
page_sha1_sig0 (segment PKCS7 signature)
page1 (second data segment)
page_sha1_1 (second segment's sha-1)
page_sha1_sig1 (segment PKCS7 signature)
additional segments
sha1 (data file sha-1)
sha1_sig (file PKCS7 signature)
End of AFF file
AFF segment header (16 bytes)
"AFF\000" (4 bytes)
length of segment name (4 bytes)
length of segment data (4 bytes)
segment flag (4 bytes)
segment name
"page0"
segment data
up to 4GB of binary data, although in practice most segments are less than 16MB
AFF segment footer (8 bytes)
"ATT\000" (4 bytes)
length of entire segment (4 bytes)

Fig. 2 – AFF file structure (left), AFF segment structure (right) (Garfinkel, 2006).

(Fig. 2, left). Each segment has its own header/footer sections with relevant length and other information (Fig. 2, right). Each segment is protected by a cryptographic hash or PKCS7 signature.

AFF is more than a format specification. It also provides a set of container-manipulation tools and a library implementation (`afflib`) with a filesystem-like API that facilitates the transition from plain files to DECs. Nonetheless, it does require the reengineering of applications and does not provide a complete solution to the auditing problem, because it does not monitor the run-time behavior of accessing applications, although it does provide the means to detect evidence-tampering through digital signatures.

Gfzip is another open-source effort, which has much in common with AFF both in terms of goals and technical approaches. In fact, there is limited compatibility between the two in the segment-based layout. However, unlike AFF, gfzip seeks to maintain compatibility with *dd*'s raw image format by allowing raw data to be placed first.

2.3. *EnCase*

DEC formats created by commercial vendors are, generally, older and technically less sophisticated. By far the most popular format is the one introduced by EnCase (formally, Expert Witness). Although some details are not well understood due its proprietary nature, it essentially consists of a header section, followed by a sequence of 32 kB compressed chunks, along with chunk CRCs. At the end, an MD5 hash of the file is attached. In addition, the file contains pointers to support fast seek operations.

2.4. *PyFlag*

PyFlag defines a forensic image format based on the seekable version of gzip called `sgzip`. The main advantage of `sgzip` is that it splits the image into fixed-sized blocks and compresses them individually to allow fast seek operations. There are no provisions for attaching metadata to the target image or to log access to the evidence. An important idea is the use of library hooks to present a filesystem view of several supported forensics formats, which is similar in spirit to our own work. However, there are no secure logging and access control methods.

2.5. *Secure audit logs*

To further strengthen the auditing capabilities for DECs, anti-tampering facilities can be introduced for the DEC contents, particularly the audit log. The goal is not to prevent tampering with the audit log or contents of a DEC, but rather to solve a slightly easier problem, to make tampering detectable. In general, secure auditing facilities require a trusted component. This component can be either a WORM (Write Once Read Many) drive, to which audit log entries are appended, or a secure server, physically inaccessible to an attacker. We discuss a few design choices.

Schneier et al. (1999) discusses one scheme for secure auditing, which involves an untrusted machine U (e.g., a machine used in a digital investigation) which shares a secret A_0 with a trusted machine T. To append a new log entry D_j, U computes $K_j = \text{hash}(A_j)$, $C = E_k(D_j)$, $Y_j = \text{hash}(Y_{j-1};\ C)$, and $Z_j = \text{MAC}_{A_j}(Y_j)$. Y_j is the jth entry in a hash chain, where

$Y_1 = 0$ and MAC is a keyed hash function. Then $[C, Y_j, Z_j]$ is written to the log. The shared secret is then recomputed: $A_{j+1} = \text{hash}(A_j)$ and A_j is destroyed. This scheme is specifically tailored to disallow log entries created before a compromise at time t from being read by an attacker. The idea is that the attacker is then left to delete the entire log (which will be noticed when communication is established in the future between U and T) or leave the log alone (and thus not know whether an entry in the log makes note of his unauthorized access). This scheme is useful if access to previous log entries by applications running on U is not needed. Note that T can verify that the audit log on U is correct, because it possesses A_0 and can ''replay'' the entire log.

Snodgrass et al. (2004) have proposed a technique that allows read access to the log while preventing widespread tampering with the audit log. The scheme makes use of a trusted notary service, which accepts digital documents, computes a hash function over the document and a secure timestamp and then stores and returns a notary ID. This notary ID is then stored with the log entry. To determine if the audit log is consistent, a trusted party can verify that the notary IDs (and associated timestamps) on the notary service match those in the audit log. Omissions, additions, and deletions can all be noticed. This basic scheme has the drawback of requiring a significant amount of communication with the notary service, but audit log entries can be combined and submitted as a single document to the notary in order to reduce communication (at the expense of a coarser level of log validation). For DEC audit logs, the Snodgrass approach is particularly attractive, since only a limited amount of storage is required on the trusted server. For each audit log entry, a hash is computed over the text of the log entry, this hash is submitted to the notary service, and the notary ID returned is then stored in the DEC's audit log. Note that the DEC's audit log is readable by any application, which is useful for creating reports, evaluating an investigation, or performing tool evaluation.

3. Forensic discovery auditing module

3.1. Design goals

The ultimate goals of the forensic discovery auditing module (FDAM) can be placed in several categories:

- *Trusted logging*. The main purpose of the FDAM is to provide a complete and trustworthy history of all data operations performed on the forensic image. Logging should be performed both at the filesystem layer and at the block-device layer to fully document all operations. Ideally, it should support various levels of detail (based on user needs) and most common formats currently in use.
- *Flexible DEC support*. It should be able to import from and export to multiple digital evidence container formats. This includes both specialized forensic containers, such as DEB and AFF, and generic ones such as plain files and raw device images.
- *Tool independence*. The module should work for any application used in the forensic process, including 'non-forensic' ones.
- *Tool/agent identification*. The module should unambiguously identify the tool used to perform each operation (along with its run-time parameters) and should be able to attribute it to the user on whose behalf the operation is executed.
- *Policy enforcement*. This is an optional requirement and is an extension of the previous one – once it is possible to identify and attribute tool use, the next step is to enable policy enforcement by blocking undesirable behavior. Policies can be based on the type of tool used (either black-list, or white-list), user identity (access control), or tool behavior (e.g., no write operations, and no block-level operations). The last option allows for the safe use of general-purpose tools that have not been tested (or certified) for forensic use.

3.2. Architecture

From a system design perspective, the logical place to install independent auditing is the operating system, specifically, the filesystem interface. Since most tools rely on the operating system components to interpret the filesystem found on the forensic image (as opposed to interpreting the raw image), installing an auditor at the gate is a logical choice. We note that, for completeness, the auditor should also be installed at the lower, block-level interface to document operations that bypass the filesystem interface. Such a discussion is beyond the scope of this article, however in Roussev et al. (2006), we have demonstrated the use of a similar block-level device. The only addition it needs is the secure logging, which would be substantially the same as with filesystem operations.

The proposed FDAM architecture is shown in Fig. 3 and operates as follows. First, DEC containers (in various supported formats) are imported by the kernel module, which results in the creation of a block-level device (raw image) of the evidence data, and a mounted filesystem (if possible), which are then presented to the applications for processing, as usual. We should point out that the import does not necessarily mean that a new physical raw image is actually created. In fact, we expect that the typical behavior would be to wrap the existing DEC with block-device/filesystem layer interfaces.

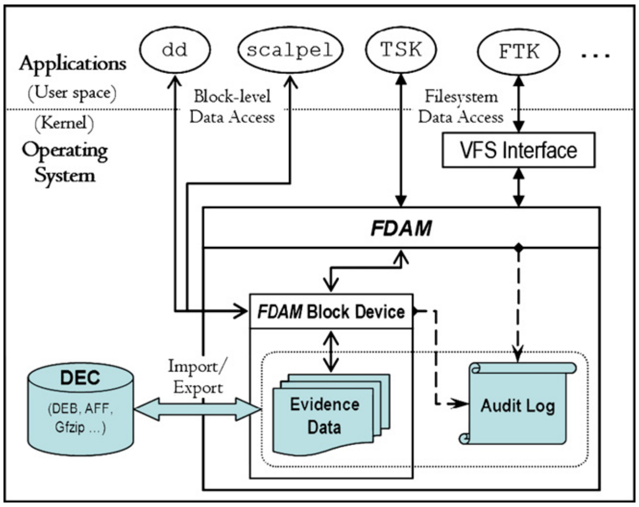


Fig. 3 – Forensic discovery auditing module (FDAM): architectural diagram.

Furthermore, FDAM presents its own API (presented later), which enables DEC-aware applications to manipulate the imported container. Once the DEC is imported, all access is monitored and logged. Finally, upon completion of the examination, the raw data and the logs are packed and sealed in a DEC of choice.

We expect that different types of DECs will be used, depending on the size and complexity of the case. Under our scheme, it becomes possible to perform and document conversions between different formats, as well as splitting and aggregation. Since all operations are automatically recorded in a trusted log, it is straightforward to demonstrate chain of custody and to verify integrity in an automated manner.

The following sections survey implementation choices for our design, define an API for DEC-enabled applications, discuss how native applications can transparently access DECs, and describe some actual experience with our prototype implementation.

3.3. *Implementation choices*

We evaluated the capabilities of several filesystems before choosing a candidate for native DEC support, including ext2/3, reiserFS, NTFS, and FUSE. NTFS was eliminated from consideration because source code is not available, although NTFS alternate data streams are an attractive mechanism for implementing DEC resource forks (e.g., blobs of digital evidence, the audit log, and DEC metadata). Most of the other filesystems we considered contain similar features that might be used to support efficient manipulation of DECs, such as support for extended attributes (EAs). But the EA support in several of the candidate filesystems is limited. In earlier work Richard and Roussev (2006), we used FUSE (File System in Userspace), to test some preliminary ideas in automatically auditing the contents of DECs. Our choice at that time was motivated by the need for rapid prototyping, but further investigation reveals that the performance of a FUSE implementation of our techniques is acceptable. But even more compelling reasons for using FUSE are flexibility and verifiability. New ideas are straightforward to implement and only a relatively small amount of additional code needs to be verified for forensic soundness. Further, the user-level FUSE code is segregated from kernel-level filesystem code. We briefly discuss FUSE before providing some technical details about our implementation.

3.4. *FUSE filesystem in Userspace*

FUSE is a system for rapid development of filesystems. The architecture of FUSE is illustrated in Fig. 4. A FUSE kernel component, which implements a Virtual File System (VFS) filesystem, traps system calls and redirects them to a user-space filesystem implementation, compiled against the FUSE library. This allows new filesystems to be quickly designed and built without the complexity of in-kernel hacking. The FUSE kernel module acts as a bridge to the VFS kernel interfaces.

To instrument system calls in FUSE, the new filesystem supplies a structure *fuse_operations* that redirects system calls through functions defined by the filesystem. This structure is depicted below.

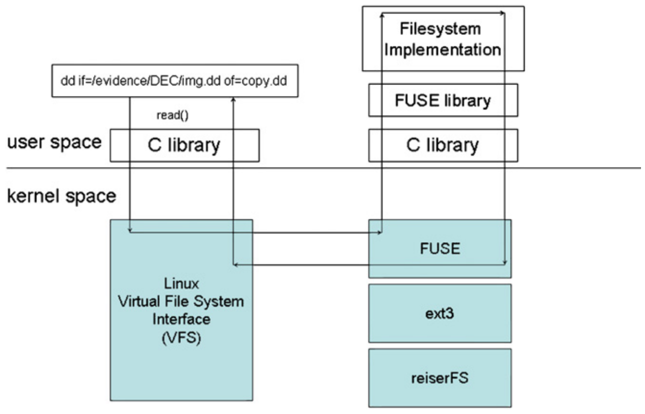

Filesystem Implementation
dd if=/evidence/DEC/img.dd of=copy.dd
read()
FUSE library
user space
C library
C library
kernel space
FUSE
Linux Virtual File System Interface (VFS)
ext3
reiserFS

Fig. 4 – FUSE architecture.

```
static struct fuse_operations aud_oper = {
  .getattr = aud_getattr,
  .access = aud_access,
  .readlink = aud_readlink,
  .readdir = aud_readdir,
  .mknod = aud_mknod,
  .mkdir = aud_mkdir,
  .symlink = aud_symlink,
  .unlink = aud_unlink,
  .rmdir = aud_rmdir,
  .rename = aud_rename,
  .link = aud_link,
  .chmod = aud_chmod,
  .chown = aud_chown,
  .truncate = aud_truncate,
  .utime = aud_utime,
  .open = aud_open,
  .read = aud_read,
  .write = aud_write,
  .statfs = aud_statfs,
  .release = aud_release,
  .fsync = aud_fsync,
#ifdef HAVE_SETXATTR
  .setxattr = aud_setxattr,
  .getxattr = aud_getxattr,
  .listxattr = aud_listxattr,
  .removexattr = aud_removexattr,
#endif
}
```

Each of these functions can completely redefine a filesystem operation, augment its functionality, or provide auditing. For example, to audit write operations, a filesystem can define the function *aud_write* to record information about the calling process and then call the standard FUSE function to carry out the write operation. The function *fuse_get_context()* allows the UID, GID, and process ID of the calling process to be obtained within each file operation. This information can then be supplemented with information from /proc (under Linux) to build the context for the write.

FUSE currently has ports for Linux and FreeBSD and a number of language bindings, including C, C++, Python, and Perl.

FUSE is being actively developed and is in widespread use; one of the most important uses of FUSE is for NTFS support in Linux. As of kernel version 2.6.14, FUSE is integrated into the Linux kernel.

3.5. *Description of prototype*

We have developed a prototype system for native filesystem support for both DEC-enabled and legacy applications, based on FUSE. Our system is developed in C and Python, under Linux. In our prototype, user-level applications are currently used to import and export DECs into and out of a special DEC-aware FUSE filesystem. An import operation essentially splits the DEC into component files and places these files into the filesystem, along with the DEC audit log and other metadata. Exporting a DEC from the DEC-enabled filesystem simply recreates the DEC structure from the data stored in the corresponding directories in the filesystem. The use of these import and export applications enables our system to be neutral with regard to developing standards for DEC structure.

Our prototype provides automatic auditing of access to DECs by both DEC-enabled and legacy applications. In the next section, we describe an API for access to DEC contents by enabled digital forensics applications. Use of the API automatically generates audit records describing how applications access the contents of a DEC. But many applications, including those specifically designed for digital forensics investigation (e.g., file carvers) and those which are not (e.g., *dd* and other common Unix command line programs) may never be modified for DEC compliance. So our implementation instruments filesystem-level system calls (through FUSE), such as file open, read, and write operations, and captures information about both the calling application and the operations themselves. Rather than using the proposed API, legacy applications may simply use the standard C library *open()*, *close()*, *read()*, and *write()* operations (and their buffered counterparts) on digital evidence blobs contained within a DEC. Legacy access to the blobs of digital evidence in a DEC automatically results in updates of the audit log. For example, an open operation records information including the user ID, process ID, MD5 hash of the accessing application's executable, the date and time, and the command line of the accessing application. Auditing of read/write operations ties these operations to the associated open operation and writes entries in the log detailing the offsets and lengths of the data accessed.

To illustrate, fragments of an audit log for a DEC containing a single disk image appear below. The contents of the DEC were accessed by a number of applications, including tools from the Sleuthkit and a file carver. The actual log is much larger; these fragments are provided to illustrate the information gathered on each operation.

```
Thu Aug 31 18:07:43 2006 uid: 0, gid: 0, pid: 22071
  eid: 1 open: /root/work/audit_fs/evidence/ext2_
image/ext2_image
  executable: /usr/bin/fsstat
  hash: 7b16b5e9abf862528f54d2405686aa27
  execution: fsstat ext2_image
Thu Aug 31 18:07:43 2006 eid: 1 read length: 32768,
starting at: 0
Thu Aug 31 18:07:43 2006 eid: 1 read length: 4096,
starting at: 65536
Thu Aug 31 18:07:43 2006 eid: 1 read length: 4096,
starting at: 262144
Thu Aug 31 18:07:43 2006 release:
  /root/work/audit_fs/evidence/ext2_image/
ext2_image
...
...
Thu Aug 31 18:07:51 2006 uid: 0, gid: 0, pid: 22083
  eid: 2 open: /root/work/audit_fs/evidence/ext2_
image/ext2_image
  executable: /usr/bin/ils-sleuthkit
  hash: 5b6e1d30a8b02d5d01adc24c4bc7b57b
  execution: ils ext2_image
Thu Aug 31 18:07:51 2006 eid: 2 read length: 32768,
starting at: 0
Thu Aug 31 18:07:51 2006 eid: 2 read length: 4096,
starting at: 65536
Thu Aug 31 18:07:51 2006 eid: 2 read length: 4096,
starting at: 262144
Thu Aug 31 18:07:51 2006 eid: 2 read length: 16384,
starting at: 32768
Thu Aug 31 18:07:51 2006 eid: 2 read length: 16384,
starting at: 49152
...
...
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072,
starting at: 940949504
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072,
starting at: 941080576
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072,
starting at: 941211648
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072,
starting at: 941342720
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072,
starting at: 941473792
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072,
starting at: 941604864
Thu Aug 31 18:07:52 2006 release:
  /root/work/audit_fs/evidence/ext2_image/
ext2_image
...
...
Thu Aug 31 18:08:04 2006 uid: 0, gid: 0, pid: 22097
  eid: 5 open: /root/work/audit_fs/evidence/ext2_
image/ext2_image
  executable: /usr/local/bin/scalpel
  hash: 8ab31b90d800ed36da88cdaa5176aaaa
  execution: scalpel ext2_image
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 0
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 131072
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 262144
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 393216
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 524288
```

```
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 655360
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072,
starting at: 786432
...
...
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072,
starting at: 744095744
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072,
starting at: 744226816
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072,
starting at: 744357888
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072,
starting at: 744488960
Thu Aug 31 18:09:32 2006 release:
  /root/work/audit_fs/evidence/ext2_image/
ext2_image
```

To provide an acceptable balance between performance and thorough auditing, our prototype allows many options to be configured on installation. One option is automatic hashing of the executables of applications accessing a DEC (e.g., when an open operation is performed). Another is automatic hashing of components of the DEC, as well as the entire DEC, after certain operations such a *write()* or a *close()*. To conclude this discussion, we note that the amount of information that can be gathered by our prototype about the calling application and the state of the DEC is enormous. Is it useful, for example, to know what other files (aside from those contained in a DEC) a calling application has open? What network connections? The version of the BIOS on the machine performing the application? Whether it has active network connections? Striking the balance between performance and adequate oversight of an investigation is an open research question.

3.6. DEC access for enabled applications

In this section we present an API for enabled applications to create, access, and modify DECs. The functions fall into three categories. The first group of functions allows digital evidence containers to be created and for "blobs" of digital evidence to be introduced into a DEC. A blob is an arbitrary unit of digital evidence and might be a disk image, a single document, or a compound file type. The second group of functions allows access to a DEC's tags. Tags record DEC metadata, such as the investigating agent's name and contact information. The third group of functions provides access to the DEC's audit log, so that applications can insert additional entries into the log to document investigative operations. Use of any of the functions automatically introduces entries into the DEC's audit log, regardless of the audit log entries entered explicitly by the enabled application. Each function in our proposed API is described briefly below.

- *int CreateDEC(char *filename, char *applicationinfo, char *comment, /* variable number of DEC tags */);*

  Creates a new digital evidence container whose complete pathname is *filename*. The *comment* field is a free-form string entered into the audit log to describe the creation event, while *applicationinfo* documents the application creating the DEC. A variable number of tags, which document the investigator's name, contact information, and case characteristics are permitted. An initial entry is made in the DEC's audit log to document the creation event. This entry also contains a hash of the initial DEC contents, which at this stage are essentially metadata. The AddDECBlob() function, described below, allows digital evidence to be introduced into the bag. A positive return value indicates successful creation of the bag.
- *int AddDECBlob(char *filename, char *blobname, void *blob, char *applicationinfo, char *comment);*

  Introduces a new piece of digital evidence, *blob*, named *blobname*, into the container whose pathname is *filename*. The blobname must uniquely identify the piece of digital evidence in the DEC, otherwise an error is generated. The *comment* is introduced into the DEC's audit log to describe the digital evidence introduced, while *applicationinfo* documents the application itself. In addition, audit log entries are automatically written to document the cryptographic hash of the introduced evidence plus a hash of the entire container contents after introduction is completed. A positive return value indicates successful introduction of the blob.
- *int AddDECBlobFile(char *filename, char *blobname, char *blobfilename, char *applicationinfo, char *comment);*

  This function performs the same operations as AddDECBlob(), except that the digital evidence to introduce is contained in the file named *blobfilename*, instead of in a block of memory.
- *int OpenDECBlob(char *filename, char *blobname, int mode, char *applicationinfo, char *comment);*

  Returns a file handle attached to the blob with name *blobname* contained in the DEC whose complete pathname is *filename*. The file handle is opened with read/write permissions described by *mode*, which has the same semantics as the mode parameter for the standard C open() function. The *applicationinfo* argument describes the application issuing the open while the *comment* describes the open operation (from the opening application's perspective) in the DEC's audit log. A positive return value indicates success.
- *void CloseDECBlob(int handle, char *comment);*

  Releases the file handle *handle*, attached to a single blob of evidence in a DEC. The *comment* describes the close operation (from the close-ing application's perspective) in the DEC's audit log.
- *unsigned long long ReadDECBlobBlock(int handle, void *data, unsigned long long len, char *comment);*

  Reads a block of *data* from the stream identified by *handle*. This handle must have been obtained from a call to OpenDECBlob(). The length of the block to read is *len*. The *comment* argument describes the read operation from the application's perspective. Returns the number of bytes read.
- *unsigned long long WriteDECBlobBlock(int handle, void *data, unsigned long long len, char *comment);*

  Writes a block of *data* to the stream identified by *handle*. This handle must have been obtained from a call to OpenDECBlob(). The length of the block to write is *len*. The *comment* argument describes the write operation from the application's perspective. Returns the number of bytes written.
- *char *GetDECTagValue(char *filename, char *tagname, char *applicationinfo, char *comment);*

Returns a pointer to a string containing the value of the tag *tagname* associated with the DEC identified by *filename*. The *applicationinfo* argument describes the application issuing the operation while the *comment* describes the operation in further detail in the DEC's audit log. NULL is returned if the tag's value cannot be returned.

- *int PutDECTagValue(char *filename, char *tagname, char *applicationinfo, char *comment);*

  Creates (or modifies) the tag identified by *tagname*, setting (or replacing) its value by *tagvalue* for the DEC identified by *filename*. The *applicationinfo* argument describes the application issuing the operation while the *comment* describes the operation in further detail in the DEC's audit log. A positive return value indicates successful modification of the tag.
- *int OpenDECAuditLog(char *filename, char *applicationinfo, char *comment);*

  Returns a file handle associated with the audit log for the DEC *filename*. The file handle's mode is read-only. This function's primary use is for reviewing the audit log. To modify the audit log, AppendDECAuditLog() must be used.
- *void CloseDECAuditLog(int handle, char *applicationinfo, char *comment);*

  Closes the audit log stream associated with *handle*.
- *int AppendDECAuditLog(char *filename, char *auditentry, char *applicationinfo, char *comment);*

  Appends a log entry *auditentry* to the audit log associated with the DEC identified by *filename*. A positive return value indicates a successful append operation.

3.7. *Performance study*

We ran a number of experiments to determine the overhead of auditing access to DECs. These experiments target unmodified legacy applications accessing blobs of digital evidence stored in a DEC inside our prototype DEC-enabled filesystem. For these experiments the Python version of the filesystem was used. The C version generally exhibits slightly better performance.

Fig. 5 presents representative performance results. We ran *cat*, *md5sum*, and *Scalpel* on a 960 MB ext2 disk image, stored in an ext3 partition and in our DEC-enabled filesystem. For each application, three executions were timed, with reboots of the test machine between each run to minimize caching effects. The average of the three runs is reported. The overhead for access to digital evidence by legacy applications in the DEC-enabled filesystem varies from 5 to 13%. Scalpel, a file carver, incurs more overhead because it makes two passes over the entire disk image. In a number of additional experiments, we have observed an average overhead of approximately 9%.

Command	ext3 FS	DEC-enabled FS
cat	59s	62s
md5sum	58s	63s
Scalpel	1m34s	1m46s

Fig. 5 – Performance results for access to digital evidence in both an ext3 filesystem and our DEC-enabled filesytem. Each application, cat, md5sum, and Scalpel, was run on a 960 MB ext2 disk image. Results are the average of three executions.

We noted in previous work (Richard and Roussev, 2006) that over Samba mounts, Windows XP (Service Pack 2) issued two parallel, non-overlapping sequences of read operations through Samba, even when application accesses were strictly sequential. Because of this anomaly, applications like FTK and Scalpel running over Samba to connect to our DEC-enabled filesystem exhibit roughly double the overhead of local applications. Additionally, accesses over Samba obfuscate the name of the application touching a blob of digital evidence. For example, if a Windows application accesses DEB data through a Samba share, the audit log shows only *smbd* as the accessing application (i.e., the Samba daemon under Linux). These limitations illustrate the need for additional work on support for networked access to DECs.

4. Conclusions

Digital forensics-aware operating system components have the potential to significantly improve the investigative process, enhancing a number of aspects, from chain-of-custody certification to testing and training.

In this paper, we explored one example, examining how introducing native support for digital evidence containers (DECs) improves auditing in an investigation. DECs mimic traditional evidence bags, by providing a standard container for arbitrary digital evidence, with an integrated audit log and metadata that describes the evidence and investigators. However, a DEC is a passive entity that cannot, by itself, log all access to the evidence – it relies on the application to provide that information voluntarily. This requires the modification of existing tools and raises the question of whether an application can be trusted to police its own actions.

Our work seeks to extend the advantages of DECs by providing a secure 'clean-room' environment where all data access to the evidence is monitored and recorded. It is implemented as an operating system module, called the forensic discovery auditing module (FDAM) that provides both a standard API and native filesystem support. Both new applications (specifically written to support DECs) and legacy applications (which use the standard Unix system calls for I/O) can take advantage of automatic auditing of forensic operations. We presented a prototype implementation based on FUSE to demonstrate the feasibility and approximate performance implications of such a module. On average, use of the DEC-enabled filesystem incurs approximately 9% overhead.

In addition to increased security and accountability in the processing of digital evidence, the FDAM forensic module can be easily adapted to support a number of additional forensics-related services. We discuss some of these below:

Trusted DEC transformations. Based on experience from other domains, it is all but certain that a variety of digital evidence container formats will be in active use at any given time. On the technical side, different formats offer different tradeoffs with respect to performance, efficiency, and presentation. More importantly, even if all vendors agreed to a standard DEC format, legacy support would still be needed. Furthermore, commercial vendors cannot afford to abandon support for their own DECs. In short, the need for DEC transformations will always exist and performing them under controlled conditions will support chain of custody procedures with an automated, audited solution.

Access control/policy enforcement. Since the forensic module can reliably identify the tools and users using it, it can be used to enforce policies dictating the use of specific (perhaps certified) tools. Alternatively, it is possible to automatically identify and deny access to tools that violate accepted behavior, such as issuing of write operations, block-level access, or access to parts of the image that have been declared (legally) off-limits. Similarly, standard access control policies can be implemented independently of (or in addition to) operating systems mechanisms. For example, Unix has a rather simplistic access control model, which makes it hard to assign specific rights to individual users.

Testing and certification. An independent auditing facility can be used to test and certify individual tools. It can be used to understand the behavior of individual tools without inspecting the source code and to pinpoint implementation errors. Another aspect of the testing process might be to evaluate how thorough a job the tool has performed, e.g., given the log, we can say whether, or not a specific block/file/directory has been accessed during the execution. Finally, all such queries on the log can be automated using standard database techniques.

Performance evaluation. Another application of the generated log is to understand application performance. For example, given two tools with similar functions, which one provides better performance? Since most forensic functions are I/O-bound, we can compare the logs generated by the tools on a benchmark target to predict relative performance. Furthermore, the data can be visualized to better understand patterns in the application's I/O behavior.

Training. The audit log can be used to demonstrate and visualize how individual tools work on the target, thereby providing more technical insight to examiners with less technical background. Similarly, it can be used to understand new application features.

5. Future work

Our system is a work in progress and an implementation suitable for public release is not yet complete. Once the initial implementation is stable, we expect to undertake a more thorough performance study and determine whether user-level filesystem enhancements offer sufficient performance, or whether modifications to an existing filesystem, such as ext3, are actually necessary.

We plan to support the import/export of the most commonly used DECs, such as EnCase and AFF. Other planned enhancements include automated audit log query and visualization, as well as report generation. Finally, we will explore a suitable mechanism for supporting and enforcing procedures and policies regarding the handling of evidence and fine-grained access control.

REFERENCES

Carrier B. Sleuthkit and autopsy, <http://www.sleuthkit.org>.

Forensics Toolkit (FTK), <http://www.accessdata.com>.

FUSE: Filesystem in User Space, <http://fuse.sourceforge.net/>.

Garfinkel S. Disk imaging with the advanced forensics format, library and tools. In: The second annual IFIP WG 11.9 international conference on digital forensics, Orlando, FL; Jan 2006.

PyFLAG: forensics and log analysis GUI, <http://pyflag.sourceforge.net/>.

Richard III GG, Roussev V. Toward secure, audited processing of digital evidence: filesystem support for digital evidence bags. In: Proceedings of the second annual IFIP WG 11.9 international conference on digital forensics, Orlando, FL; 2006.

Roussev V, Richard III GG, Tingstrom D. dRamDisk: efficient ram sharing on a commodity cluster. In: Proceedings of the 25th international performance computing and communications conference (IPCCC), Phoenix, AZ; Apr 2006.

Schneier B, Kelsey J. Secure audit logs to support computer forensics. Proceedings of ACM Transactions on Information and System Security May 1999;2(2).

Snodgrass R, Yao SS, Colberg C, Tamper detection in audit logs, In: Proceedings of the 30th VLDB conference, Toronto; 2004.

The generic forensic zip project, <http://www.nongnu.org/gfzip/>.

The Linux NTFS project, <http://www.linux-ntfs.org/>.

Turner P. Unification of digital evidence from disparate sources (digital evidence bags). In: Proceedings of the fifth annual digital forensics research workshop (DFRWS), New Orleans, LA; Aug 2005.

Turner P. Selective and intelligent imaging using digital evidence bags. In: Proceedings of the sixth annual digital forensics research workshop (DFRWS), Lafayette, IN; Aug 2006.