Digital Forensics XML and the DFXML Toolset

Simson Garfinkel

Naval Postgraduate School, 900 N. Glebe, Arlington, VA 22203

1. Introduction

Digital Forensics XML (DFXML) is an XML language designed to represent a wide range of forensic information and forensic processing results. By matching its abstractions to the needs of forensics tools and analysts, DFXML allows the sharing of structured information between independent tools and organizations. Since the initial work in 2007, DFXML has been used to archive the results of forensic processing steps, reducing the need for re-processing digital evidence, and as an interchange format, allowing labeled forensic information to be shared between research collaborators. DFXML is also the basis of a Python module (`dfxml.py`) that makes it easy to create sophisticated forensic processing programs (or "scripts") with little effort.

Forensic tools can be readily modified to emit and consume DFXML as an alternative data representation format. For example, the PhotoRec carver (Grenier, 2011) and the MD5DEEP hashing application (Kornblum, 2011) were both modified to produce DFXML files. The DFXML output contains the files identified, their physical location within the disk image (in the case of PhotoRec), and their cryptographic hashes. Because these programs now both emit compatible

Email address: `slgarfin@nps.edu` (Simson Garfinkel)

DFXML, their output can be processed by a common set of tools.

DFXML can also document provenance, including the computer on which the application program was compiled, the linked libraries, and the runtime environment. Such provenance can be useful both in research and in preparing courtroom testimony.

DFXML's minimal use of XML features means that the forensic abstractions, APIs and representations described in this paper can be readily migrated to other object-based serializations, including JSON (Zyp and Court, 2010), Protocol Buffers (Google, 2011) and the SQL schema implemented in SleuthKit 3.2 (Carrier, 2010). Indeed, it is possible to readily convert between all four formats.

1.1. The need for DFXML

Today's digital forensic tools lack *composability*. Instead of being designed with the Unix approach of tools that can be connected together to solve big problems, most commonly used forensic tools are monolithic systems designed to ingest a small number of data types (typically disk images and hash sets) and produce a limited set of output types (typically individual files and final reports). This is true both of tools with limited functionality (e.g. simple file carvers), as well as the complex GUI-based tools that include integrated scripting languages. The lack of composability has complicated automation and tool validation efforts, and in the process has subtly limited the progress of digital forensics research.

Although there are existing file formats and a few XML languages used in digital forensics today, they are confined to specific applications and limited domains. The lack of standardized abstractions makes it difficult to compare results produced by different tools and algorithms. This lack of standardization similarly impacts tool developers, who must frequently implement functions in their tools redundant with functions already existant in other tools.

1.2. Specific uses for DFXML

DFXML improves composability by providing a language for describing common forensic processes (e.g., cryptographic hashing), forensic work products (e.g., the location of files on a hard drive), and metadata (e.g., filenames and time stamps).

Various prototype DFXML implementations have been used by the author since 2007 for a variety of purposes:

- A tool based on SleuthKit called *fiwalk* (§5.1) ingests disk images and reports the location and associate file system metadata of each file in the disk image. This tool was used by students for Masters' theses (Migletz, 2008; Huynh, 2008), and a project that applied machine learning to computer forensics (Garfinkel, Parker-Wood, Huynh, and Migletz, 2010).
- A DFXML file was created for each disk image in a corpus of more than 2000 disk images acquired around the world (Garfinkel, Farrell, Roussev, and Dinolt, 2009). Each DFXML file contains information regarding the disk's purchase, physical characteristics, imaging process, allocated and deleted files, and metadata extracted from those files (e.g., Microsoft Office document properties, extracted JPEG EXIF information, etc.).
- The DFXML Python modules (§4.1) make it possible to write small programs that perform complex forensic processing on these files (**?**). In contrast, the learning curve for tools such as EnCase EnScript (Guidance Software, 2007) and SleuthKit (Carrier, 2010) can be quite steep.
- The XML files make it dramatically easier to share data with other organizations. In some cases it has only been necessary to share the XML files,

rather than the disk images themselves. This is more efficient, as the files are much smaller than the disk images. It also helps to protect the privacy of the data subjects.

- The XML format makes it easy to identify and redact personal information. The resulting redacted XML files can be shared without the need for Institutional Review Board (IRB) or Ethics Board approval; they can even be published on the Internet.
- Finally, because the DFXML files record which version of which tool produced each file, it is easy to have tools automatically reprocess disk images when the toolset improves.

1.3. Contributions

This paper makes several specific contributions to the field of digital forensics. First, it describes the motivation and design goals for DFXML. Second, the paper presents specific examples of how DFXML can be used to describe forensic artifacts. These examples make it easy for developers of today's forensic tools to adopt their tools to emit and ingest DFXML as a complement to their current file formats. Next, it presents an API that allows for the rapid prototyping and development of forensic applications. Finally, it describes how the DFXML abstractions can be used as a building block for creating new automated forensic processes.

2. Prior Work

Although file formats, abstractions, and XML are all used in digital forensics today, they are rarely themselves the subject of study. Mainly, these topics arise when practitioners discover that they cannot share information with one another, or even between different tools, because data are stored in different formats.

2.1. Digital evidence containers

Broadly speaking, *digital evidence containers* are files designed to hold digital evidence. Most common are disk image files that hold sector-for-sector copies of hard drives and other mass storage devices. The simplest disk image is a raw format (also called *dd format* after the Unix `dd` program).

Modern disk image formats can use lossless compression and de-duplication to decrease the amount of storage space required, while still allowing the recreation of the original disk image. Although disk image formats such as Norton Ghost, VMWare VMDK, Apple DMG and Microsoft WIM have been used for years within the IT community, forensic practitioners have mostly standardized on the Expert Witness Format (EWF) used by Guidance Software's EnCase program. (The format is also known as the *.E01* format after the file extension.) EWF includes limited support for representing metadata such as the date that a disk image was acquired and the name of the examiner who performed the acquisition, as well as a free-format "notes" field, but does not support the representation of structured forensic information.

Kloet, Metz, Mora, Loveall, and Schreiber (2008) presented an open source implementation of EWF in C; Allen (2009) presented an EWF implementation in Java (a C# implementation is also under development). These open source implementations make it possible to read any sector of a disk image in EWF format and also make it possible to read the limited metadata that accompanies the disk image. Of course, these implementations must be combined with software such as SleuthKit in order to extract individual files from the disk image.

Turner proposed a "wrapper" or metaformat called "Digital Evidence Bags" (DEB) to store digital forensic evidence from disparate sources (Turner, 2005). The DEB consists of a directory that includes a *tag* file, one or more *index* files, and one or more *bag* files. The tag file is a text file that contains metadata such as the name and organization of the forensic examiner, hashes for the contained information, and data definitions. Turner created a variety of prototype tools, including a Digital Evidence Bag Viewer and a Selective Imager.

Cohen, Garfinkel and Schatz introduced AFF4 (Cohen, Garfinkel, and Schatz, 2009), a redesign of Garfinkel's Advanced Forensic Format (AFF) (Garfinkel, 2006). Both AFF and AFF4 store disk images and associated metadata. AFF4 uses a flat RDF schema to store this auxiliary information. Although the RDF schema can be used to store file and filesystem metadata, this is not frequently done in practice, and tools to create such RDF files are not generally available.

2.2. Representing Registry information

There has been considerable forensic research aimed at recovering allocated data from Windows Registry hive files (Howell, 2009) and from unallocated space inside the hive (Thomassen, 2008; Tang, Ding, Xu, and Xu, 2009).

Because of limitations of the ASCII-based registry file format defined by Microsoft's RegEdit tool, several developers created tools for extracting Registry entries from hive files and representing the resultant information as XML(Rodriguez, 2003; Shayne, 2001; Jones, 2009).

Finally, the National Institute of Standards and Technology's WIRED project has developed a program called *reg-diff.rb*, which ingests two ASCII files generated by RegEdit and produces an XML file describing the differences (Dima, 2006).

2.3. File system metadata standards

File system metadata is the name given to information within a file system other than file contents, including file names, timestamps, access control lists and disk labels. File system metadata is widely used in computer forensics as the primary tool for navigating file system information and reconstructing event timelines.

To date there has been little effort to develop standard descriptions of file system metadata. The Coroner's Toolkit (Farmer and Venema, 2005) introduced a "body file" format containing 16 entries for each file including file name, size, MAC times, allocation status, and other metadata that can be recovered from a file system. Individual fields were separated by pipe symbols (|) to allow for easy parsing by programs written in Perl. Body files were designed for moving data from one tool to another in the Toolkit, but not for data archiving or exchange between examiners. Carrier preserved the file format in SleuthKit 2.0 but modified it in SleuthKit 3.0 by reducing the number of fields to 11, rendering old files incompatible with the new tools and vice-versa.

2.4. File metadata and extracted features

The Electronic Discovery Reference Model (EDRM) XML (Socha, 2011) is a data interchange format for describing metadata of interest to e-discovery practitioners, including the Microsoft proprietary metadata fields embedded within Word and PowerPoint office files, and the *To:*, *From:* and *Subject:* fields of email messages. EDRM XML does not describe the physical location of a file on a hard drive or the MD5 hash of individual sectors.

The National Information Exchange Model is an effort by the US Department of Justice, the US Department of Homeland Security, and the US Department of

Health and Human Services to create standardized data models for the sharing of structured information between different federal agencies. Of interest to forensics practitioners is the Terrorist Watchlist Person Data Exchange Standard, which provides a schema for describing identity information (US Department of Justice and US Department of Homeland Security, 2011).

2.5. XML Languages for computer security

Frazier (2010) of MANDIANT developed Indicators of Compromise (IOCs), an XML-based language designed to express signatures of malware such as files with a particular MD5 hash value, file length, or the existence of particular registry entries. There is a free editor for manipulating the XML. MANDIANT has a tool that can use these IOCs to scan for malware and the so-called "Advanced Persistent Threat."

MITRE's "Making Security Measurable" program has developed three XML languages for describing items of importance to computer security practitioners and researchers. The MITRE project consists of the Open Vulnerability and Assessment Language (OVAL®), the Common Event Expression (CEE^{TM}), and the Malware Attribution Enumeration and Characterization (MAEC^{TM}) languages.

Both MANDIANT's IOC and MITRE's MAEC are similar to DFXML in that they can describe file names, file system properties such as paths, permissions, and modification dates, and hash values. Both are able to express items not envisioned by DFXML; IOC can even contain conditional logic. But both lack the ability to express a variety of features of forensic interest, including hash values that correspond to specific byte runs within an object, the ability to specify the physical location on a piece of media, and the ability to specify a variety of file system attributes such as allocation status.

2.6. XMLs for media forensics

There has been limited work developing XML languages specifically for digital forensics.

Alink *et al.* presented XIRAF (an XML Information Retrieval Approach to digital forensics) at NLPXML 2006 (Alink, Jijkoun, Ahn, de Rijke, Boncz, and de Vries, 2006b) and DFRWS 2006 (Alink, Bhoedjang, Boncz, and de Vries, 2006a). The authors stressed the importance of having "a clean separation between feature extraction and analysis" and the importance of having "a single, XML-based output format for forensic analysis tools." XIRAF stores XML documents in an XML-aware database; examiners conduct forensic investigations through the use of XML queries.

Levine and Liberatore (2009) presented DEX (Digital Evidence Exchange) at DFRWS 2009; DEX had the goals of making it possible to reproduce the original evidence from the XML description, and of enabling tool comparison and validation. DEX made extensive use of XML attributes that required complex parsing rules. The authors released a DEX tool written in Java under a BSD-like license.

Grenier designed a XML log file for the PhotoRec (Grenier, 2011) carver. Grenier did not implement his original design, but instead graciously accepted patches from the author of the present article and incorporated DFXML into PhotoRec 6.12.

Finally, Carrier developed `tsk_loaddb`, a SleuthKit component that extracts metadata from a disk image and stores it in an SQLite database (Carrier, 2010). This approach makes it easy for those familiar with SQLite to work with digital forensics data.

3. Digital Forensic Abstractions and Digital Forensics XML

Today the most common ways for forensic practitioners to exchange forensic data are disk images and text files. For example, an investigator might give an analyst a disk image of a captured USB drive and an ASCII list of MD5 hash values and ask the analyst if any of the files in the list are on the drive. Although this approach works in practice, it does not lend itself to evolutionary growth. For example, there is no standard way to annotate that list of MD5 codes with SHA1 values, similarity digests, or classification levels. Instead, every person that wishes to annotate a list needs to develop their own *ad-hoc* format, and every tool that would interpret such a list needs to be able to handle such formats. Analysts, most of whom cannot program, spend a lot of time in Microsoft Excel adding and removing columns to overcome the diversity of formats that have evolved in recent years.

Other areas of information technology have successfully outgrown similar exercises in babble. For example, the growth of the World Wide Web is often attributed to the development of the HTML and HTTP standards, which made it possible for different groups to write software that interoperated without prior arrangement or the need for pairwise engineering efforts. Clearly, the Web also owes its birth to POSIX, TCP/IP, and the Berkeley Sockets API.

Digital forensics can similarly benefit from standardized abstractions, representations and interfaces. Such abstractions can leverage existing concepts and further enable digital forensics processes, allowing tools, practitioners and organizations to communicate more efficiently about forensic processes, while simultaneously providing an evolutionary path to exchanging increasingly sophisticated representations.

3.1. Example 1: Using DFXML to describe file locations

Consider a JPEG file on a FAT32 SD card. Agreed upon abstractions, conventions and standards allow the SD card to be moved from a digital camera to a PC running Windows or a Macintosh running MacOS. These computers can use the same name to access the same sequence of bytes that make up the JPEG file, and when desktop computers display the file on their computer screens, the pictures look virtually indistinguishable.

Forensic tools do not enjoy the same level of interoperability when it comes to describing deleted JPEGs or carving artifacts that might be found on the same SD card. The only way to determine if a specific deleted file recovered by SleuthKit and EnCase are the same is to compare the files byte-by-byte or to compare the sector numbers from which the deleted files were recovered. Other approaches, such as comparing hash values of the two files, may not be satisfactory. Many forensic examiners still perform such comparisons using the MD5 hash algorithm, and there are now multiple documented cases of different files that have the same MD5 hash value (Diaz, 2005; Selinger, 2009; Microsoft, 2008). Another disadvantage of using hash value comparision is that file similarities may be inadvertantly obscured. This can happen because the length of a carved file cannot be unambigiously determined. If two carvers identify the same file with the same starting point but the lengths are off by one byte, a hash value comparision will report that the files are different, while a byte-run comparision will report that one file is a subset of the other.

File systems have an advantage over forensic tools: Whereas standards and convention clearly define the mapping between an allocated file and a set of disk blocks, "undelete" is not a well-defined operation. Different tools undelete differ-

```
<byte_runs>
  <byte_run offset="0"     img_offset="114688"  len="32768"/>
  <byte_run offset="32768" img_offset="1523712" len="32768"/>
  <byte_run offset="65536" img_offset="6356992" len="39659"/>
</byte_runs>
```

Figure 1: Each *byte_run* XML tag specifies a mapping of logical bytes in a file to a physical location within a disk image. They can be combined in the *byte_runs* tag to specify fragments of a fragmented file.

ently, because the information on the hard drive required to perform the undelete operation may be incomplete, ambiguous, or contradictory. CarvFS attempts to solve this problem through the use of file names that are interpreted by the file system as pointers to specific disk blocks (Meijer, 2011). But CarvFS is limited to representing the location of the data on the drive—attempts to encode other information in the file names would result in prohibitively long names, and such encoding would ultimately result in names with structured attributes similar to what has been developed for DFXML.

An alternative approach employed by DFXML is to create a high-level language for describing where on a disk a file's content resides within a forensic disk image. For example, a JPEG file split into three pieces can be described as a set of three byte runs, each with a logical offset within the file, a physical offset within the disk image, and a length, as shown in Figure 1.

The *byte_run* approach is readily extended to describe logical byte runs that are zero-filled (and thus do not appear on the physical media) by replacing the `img_offset=` attribute with a `fill="0"` attribute. Likewise, NTFS compression is represented with the attributes `transform="NTFS_DECOMPRESS" raw_len="155"`.

DFXML expresses all sizes and extents in bytes, as runs do not necessarily

```
<fileobject>
  <filesize>105195</filesize>
  <partition>1</partition>
  <ALLOC>1</ALLOC>
  <crtime prec="2">2008-12-25T04:21:44</crtime>
  <mtime prec="2">2008-12-25T04:21:44</mtime>
  <atime prec="86400">2008-12-24T00:00:00</atime>
  <filename>DCIM/100CANON/IMG_0044.JPG</filename>
  <libmagic>JPEG image data, EXIF standard 2.2</libmagic>
  <byte_runs>
    <byte_run offset="0"     fs_offset="88576"   img_offset="114688" len="32768"/>
    <byte_run offset="32768" fs_offset="1497600" img_offset="1523712" len="32768"/>
    <byte_run offset="65536" fs_offset="6330880" img_offset="6356992" len="39659"/>
  </byte_runs>
  <hashdigest type="md5">cef79634dd3a86455a2cd900a691adf3</hashdigest>
  <hashdigest type="sha1">916a88a00c58b7a566711acd25e61d549df5d303</hashdigest>
</fileobject>
```

Figure 2: The completed `<fileobject>` XML element for *IMG_0044.JPG*. Notice that the create and modify times are accurate to two seconds, while the access time is only accurate to one day. All times are given without a UTC offset, since FAT32 file systems store time in local time. (Linebreaks and pretty-printing added for legibility.)

start on sector boundaries (for example, small NTFS files are resident within the MFT) and because the sector numbers cannot be interpreted without knowing the sector size—extrinsic information that may be missing or incorrect.

It is straightforward to modify existing programs to generate the `<byte_runs>` tag. Once these modifications are made, it is trivial to compare the output of different versions of a program for regression testing, or to compare the results of processing the same data with different tools for conformance testing and certification.

The complete `<fileobject>` element for the JPEG in question (taken from Garfinkel et al. (2009)) appears in Figure 2.

3.2. Example 2: Using DFXML for hash lists

While today it is common to distribute a set of file hashes as a tab-delimited file containing file names and MD5 hash values, a DFXML file of hashes can be expanded to include SHA1 and SHA256 hashes, descriptions of each file, classification levels, partial hashes of key sectors, and even the email address of an individual who should be contacted if the file is encountered. The use of XML means that adding such fields does not impact older programs that do not expect such data. As such, DFXML makes it possible to gradually evolve interchange formats, giving researchers and practitioners the ability to put increasingly sophisticated analysis results or new annotations in their interchange and archive files.

3.3. Goals for DFXML

Previous efforts aimed at developing new formats and abstractions designed for computer forensics have largely failed. For example, DFRWS launched a project in 2007 to create standardized abstractions for digital forensics; this project was abandoned within a year due to the lack of support and funding (Common Digital Evidence Storage Format Working Group, 2007). Based on the DFRWS experience, it seems reasonable that any effort to create an XML language for digital forensics should be envisioned as a low-cost project that nevertheless can produce significant savings or provide new capabilities. The following goals are compatible with such financial realities:

1. **Complement existing forensic formats.** Rather than replacing existing formats, the new language should augment them. This is accomplished by making it easy to convert between legacy and new formats, and by developing techniques so that the new formats can be used to annotate legacy

data.

2. **Be easy to generate.** It must be *easy* to modify existing tools to generate the new representations. An open source C and C++ library aids in the modification process.
3. **Be easy to ingest.** Likewise, it must be *easy* to modify existing tools to read and process DFXML. An open source DFXML Python module based on the Python SAX XML parser makes it possible to efficiently read and process DFXML files (see Section 4).
4. **Provide for human readability.** A forensic analyst with no training should be able to look at a conforming DFXML file and make sense of it without the need for a special viewer. To this end, many tools produce DFXML that is pretty-printed.
5. **Be free, open, and extensible**. Both the representation and reference implementation must available for all to use, without a license fee. Developers should be able to add new tags without the need for central coordination (accomplishable through the use of XML namespaces).
6. **Provide for scalability.** The representation must be usable at both ends of forensic scale. Small amounts of information must have short descriptions, while it must be possible to efficiently process XML documents tens of gigabytes in size (which might result from processing multi-terabyte drives). As such, it must be possible to process DFXML using event-based XML parsers (e.g., Python Software Foundation (2010); Cameron, Herdy, and Lin (2008); Zhang and van Engelen (2006)), rather than requiring the use of tree-based parsers such as those based on the Document Object Model.
7. **Adhere to existing practices and standards.** Where possible, DFXML

should follow existing standards rather than inventing new ones. Where multiple, conflicting standards exist, DFXML should implement these standards that are the most efficient and appropriate for forensic processing.

3.4. Overall design

DFXML is intended to represent the following kinds of forensic data:

- Metadata describing the source disk image, file, or other input. Typically this is the name of the image file, but may include other information.
- Detailed information about the forensic tool that did the processing (e.g., the program name and version number, where the program was compiled, linked libraries).
- The state of the computer on which the processing was performed (e.g., the name of the computer; the time that the program was run; the dynamic libraries that were used).
- The evidence or information that was extracted, how it was extracted, and where it was physically located.
- Cryptographic hash values of byte sequences.
- Operating-system-specific information useful for forensic analysis.

Each type of data are represented by a family of XML elements:

`<creator>` The program that created the XML file.

`<volume>` A mass storage system *volume*, which is defined as a collection of byte blocks that are all the same size (e.g., a hard drive, a partition within a hard drive, a RAID volume, etc.).

`<fileobject>` A file, which is a sequence of bytes with associated metadata.

`<byte_run>` A specific location of bytes on a mass storage device. These can be grouped in a `<byte_runs>` array.

`<hashdigest>` Represents a cryptographic hash.

`<msregistry>` One or more Microsoft Windows Registry entries.

DFXML also adopts by reference these two platform-specific families of XML elements:

`<database>` An SQL database, using the XML format produced by MySQL's *mysqldump* command.

`<plist>` Apple Macintosh property list information, using the XML format produced by Apple's *plutil*.

`<kml>` Geospatial information in KML format.

Although it is tempting to combine the `<database>` and `<plist>` tags into a single platform-independent schema, there is little need to do so; any processing would necessarily be done with programs that are themselves specific to a particular program that generated the data.

3.5. Combining elements to express complex concepts

DFXML elements from different domains can be combined to improve the expressiveness of the language. For example, the `<hashdigest>` element can be used to describe hashes, as shown in Figure 4. But the `<byte_runs>`, `<run>` and `<hashdigest>` elements can also be combined to describe piecewise hashing of any file or string, as shown in Figure 5. Likewise, Dublin Core Metadata Initiative (2010) annotations can be used to describe entire disk images, individual files, or even byte runs within a file.

This flexibility allows the same XML representation to be used for a variety of purposes. For example, *fiwalk generates* a DFXML structure containing a set of `<fileobject>` elements that denote the location in a disk image of specific files (**?**). In such a DFXML file, the `<fileobject>` elements have absolute pathnames

```
<dfxml version="1.0">
  <metadata
  xmlns="http://www.forensicswiki.org/wiki/Category:Digital_Forensics_XML"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
    <dc:type>Disk Image</dc:type>
    <dc:publisher>Naval Postgraduate School, Monterey, CA</dc:publisher>
    <dc:abstract>32MB SD card from a Canon Digital Camera</dc:abstract>
    <dc:date>2010-02-04T10:32:10-0800</dc:date>
  <classification>UNCLASSIFIED</classification>
  </metadata>
  <source>
   <image_filename>nps-2009-canon2-gen6.raw</image_filename>
   <acquisition_date>2010-01-02T09:30:200-0800</acquisition_date>
   <sectorsize>512</sectorsize>
  </source>
</dfxml>
```

Figure 3: Dublin Core Metadata Initiative tags can be used to annotate DFXML objects, as shown here. The schema can also be extended—for example, by including a new tag to denote the security classification of the disk image.

```
<hashdigest type="md5">e4d7f1b4ed2e42d15898f4b27b019da4</hashdigest>
<hashdigest type="sha1">b7e23ec29af22b0b4e41da31e868d57226121c84</hashdigest>
<hashdigest type="sha256"
            base="64">Ccp+TqpuiunH0mEWcSkYSINkTQffuny/vEyKLgg2DVs=</hashdigest>

```

Figure 4: Three hashes for the same string, showing how hashes can be represented as hex or base64 numbers. (Base64 representations are allowed for brevity but of course should never be entered from within a user interface.) All of these hashes are for the same sequence of 12 bytes, “hello, world”.

```
<byte_runs>
  <byte_run offset='0' len='6'>
    <hashdigest type='md5'>614eef5f3ec073b9cc4c09d211e275aa</hashdigest>
  </byte_run>
  <byte_run offset='6' len='6'>
    <hashdigest type='md5'>b7913aa15c43be7d534b4eec6e99e8a0</hashdigest>
  </byte_run>
</byte_runs>
```

Figure 5: The *byte_runs*, *run* and *hashdigest* tags can be described to denote piecewise hashing of any object. Here the first MD5 hash is for the characters "hello," while the second sequence is for the space and the letters "world."

based in the root of the file system in which they are found. As mentioned above, the popular PhotoRec carving tool now also produces DFXML files. However the DFXML produced by PhotoRec contains not the names of the files in the disk image, but instead the names of the files output by the carver; here, the file names are relative to the directory in which PhotoRec's DFXML file is written. Likewise, the DFXML files produced by *md5deep* embed absolute pathnames by default, but will contain relative pathnames if *md5deep* is invoked with the "-r" flag.

Having both filesystem extraction tools and file carvers produce the same XML makes it possible to create a forensic processing pipeline that preserves semantic content while allowing later stages of the pipeline to be insensitive to the manner in which data was extracted by the earlier stages. Having a single format supported by multiple carvers makes it possible to cross-validate carvers; to build a single "meta" file carver that logically combines the results of multiple carvers; and to perform regression tests.

3.6. Times, dates and durations

The representation of times, dates and durations is an enduring problem in information technology due to the interplay of cultural norms, the range of values that must be represented, daylight savings time, and even variances in the rotation of the earth. An added complication in digital forensics is that some legacy time representations are in local time and cannot be converted to an absolute time without the use of extrinsic information. For example, times in the Microsoft FAT32 file system are stored in local time; arbitrarily assigning these times to a specific UTC offset frequently introduces errors in the analysis process.

3.6.1. Choice of Representation: ISO 8601

There are two competing approaches for representing time in modern computer systems. One is to record the number of seconds from an epoch and to convert this value to a printable local time as needed. Unix uses this approach with an epoch of January 1, 1970 GMT; Windows uses the same approach, although with an epoch of 1601, the first year of the Gregorian calendar. Absolute time of less than a second can be represented using floating-point time (as is done in the Python programming language), or using integer time units less than a second (Windows uses nanoseconds).

The second approach is to represent time as a printable string that must be parsed. This is the approach used by the ISO 8601 standard (ISO, 2000).

The epoch-based approach minimizes storage requirements (timestamps from 1902 through 2038 can be can be stored with a single 32-bit signed integer) and simplifies many calculations. However the epoch-based approach has multiple disadvantages which make it inappropriate as a general interchange format for digital forensics, including:

1. The epoch is typically based in GMT and must be converted to a local time zone. As such, this approach cannot directly represent a "local" time for which the UTC offset is unknown.
2. Epoch-based timestamps are not capable of representing leap seconds, since future leap seconds are not known in advance. The POSIX standard actually requires that leap seconds be ignored, justifying this decision: "Not only do most systems not keep track of leap seconds, but most systems are probably not synchronized to any standard time reference. Therefore, it is inappropriate to require that a time represented as seconds since the Epoch precisely represent the number of seconds between the referenced time and the Epoch" (IEEE, 2004) Currently, leap seconds, when they occur, are represented as an extra second during the last minute of June 30th or December 31st; 2008-DEC-31T23:59:60 was most recent leap second. Hack, Meng, Froehlich, and Zhang (2010) discussed the problem of leap seconds as they apply to epoch-based time reprensetations in detail.
3. Epoch-based timestamps assume that the system to which they refer properly understand rules for daylight savings time—rules that are complex and subject to change. Indeed, there is no way to use an Epoch timestamp to represent the time on a computer whose operating system does not properly follow DST rules, or whose clock is set to the wrong timezone, without external information.
4. There are four different APIs for programmatically representing Unix timestamps: integer *time_t* values, used in legacy C programs; the *timeval* structure, which provides microsecond resolution; the *timespec* structure, which provides nanosecond resolution; and floating point timestamps, popularized

by the Python programming language. As a result, writing portable forensic software that can properly process time with sub-second resolution can be challenging.

ISO 8601 has the advantage of unambiguous representation and the ability to represent any date, time, or date and time combination. The primary disadvantage is a higher storage overhead (20 bytes instead of 4 to represent timestamps with one-second resolution prior to 2038) and higher computational overhead to ingest and emit (although some of this overhead can be negated by keeping timestamps as strings within programs).

Based on this analysis, DFXML uses ISO 8601, and specifically the WC3 ISO 8601 XML Schema (Biron and Malhotra, 2004), to represent all time values, with these addenda:

- RFC 3339 specifies a "profile" or restrictive subset of ISO 8601. Where possible, this profile should be used by DFXML implementations.
- Time precision or resolution is specified in seconds using the XML attribute `prec=`. For example, FAT32 create and modify times are accurate to 2 seconds but access times are only accurate to 1 day. When not present, precision is assumed to be 1 second.
- Time values with sub-second precision are represented as floating point seconds. For example, 1 nanosecond after midnight, January 1, 2010 is specified as `2010-01-01T00:00:00.000000001`.
- Strict adherence to the ISO 8601 standard requires durations ("periods") to be expressed with strings such as `P3600S` rather than simply as `3600`. However, ISO 8601 allows the same duration to be expressed as `P1H` or `PT60M`. This ambiguity has the effect of increasing the complexity of parsing, violat-

ing Goal 3. As such, durations in DFXML are always expressed as floating point seconds.

3.6.2. Performance

Although the ISO 8601 representation requires more computational effort than epoch-based timestamps to emit and ingest, the extra overhead is not significant.

An ISO 8601 parser written in C based on the standard C library *strptime* function achieves nearly 260,000 conversions per second on a 2.26Ghz Intel processor. The same hardware can perform 26 million string-to-integer conversions per second, making ISO 8601 parsing two orders of magnitude slower. Nevertheless, timestamp parsing is not likely to be the most computationally burdensome aspect of processing a large DFXML file. Amdhal's Law suggests that optimization efforts are better spent elsewhere.

It is instructive to note that it is more efficient to parse ISO 8601 timestamps in C, rather than using higher-level parsing functions provided by languages such as Python. For example, Python's native *datetime* parser can perform only 6100 conversions from ISO 8601 to *time_t* each second on the same hardware. Thus, rather than using Python's *datetime* parser, it is better to use Python's *strptime* function, which merely calls the corresponding function in the C library.

3.7. Windows Registry

It is useful to have a means for representing specific collections of Microsoft Windows Registry entries to describe the installation and behavior of applications, the results of file carving, and even the behavior of attackers.

Although multiple approaches have been created for representing registry entries in XML, no approach is in widespread use. DFXML therefore uses a representation loosely based on Shayne (2001), but with the tags in lowercase for

```
<msregistry>
  <key name="HKEY_CURRENT_USER" root="1">
    <key name="Console">
      <mtime>2010-03-24T10:10:10-04:00Z</mtime>
      <value name="ColorTable00" type="REG_DWORD" value="0"/>
      <value name="ColorTable01" type="REG_DWORD" value="8388608"/>
...
      <value name="WindowSize" type="REG_DWORD" value="1638480"/>
      <value name="WordDelimiters" type="REG_DWORD" value="0"/>
    </key>
  </key>
</msregistry>
```

Figure 6: An example of RegXML, the XML representation used by DFXML to represent registry entries. The `root="1"` attribute indicates that this key starts at the registry's root. Of course, this example could be updated to include provenance information regarding the tool that created the entry.

consistency with the other DFXML tags.

Each key in the Windows Registry contains a Windows 64-bit timestamp denoting the last time it was modified (Morgan, 2009). DFXML represents this last write time through the use of an `mtime` element with the `<key>` tag (Figure 6).

The `<byte_run>` element can be used to annotate any `<key>` or `<value>` to indicate the physical location that the value was found. This is useful when reconstructing orphan registry tags found in unallocated regions of the Windows registry hive or in memory (Figure 7).

Although it is possible to extract the entire Windows registry as a single XML document, it is rarely useful to do so. Instead, XML is useful for representing specific registry settings that have been extracted and for representing templates or rules.

```
<msregistry version="1.0">
  <key name="Media Center">
      <mtime>2010-04-21T15:23:41-04:00Z</mtime>
      <byte_run offset="23423450" len="932"/>
      <value name="" type="REG_SZ" value=""/>
      <value name="10 Foot OOBE" type="REG_DWORD" value="0">
         <byte_run offset="234211320" len="532"/>
      </value>
      <value name="EnableSPADLaunchErrors" type="REG_DWORD" value="1">
        <byte_run offset="234214440" len="332"/>
      </value>
      <value name="Ident" type="REG_SZ" value="6.0"/>
  </key>
  <value name="Item 1" type="REG_SZ"
         value="[F00000000][T01CC0513A459AE40][O00000000]*\Documents\Job Report.docx">
         <byte_run offset="33421020" len="1432"/>
  </value>
  <value name="AES Key" type="REG_BINARY"
         value="233cfdcba7901ef231f646e9ad06f40fcbe62d5172255fe49097cd751ed83480">
         <byte_run offset="8987332" len="1001"/>

  </value>
</msregistry>
```

Figure 7: This example of RegXML shows how unallocated key/value pairs found within a registry hive can be represented. In this case, an orphaned Media Center registry key was found 23423450 bytes into the registry hive, an orphaned value from a Most Recently Used (MRU) list inside Microsoft Word was found at location 33421020, and a value claiming to be an AES key found at offset 8987332.

3.8. Provenance

In addition to storing information about the forensic object being analyzed, it is frequently useful to include information about the specific tools used to create the XML file. In DFXML, this provenance information is indicated with a `<creator>` element that includes data about how the tools used to generate the XML were compiled and how they were run (Figure 8).

3.9. Metadata annotations with DCMI

The data dictionary developed by Dublin Core Metadata Initiative (2010) can be readily used to annotate both entire DFXML files (for example, to provide an abstract for a disk image), or to annotate specific elements within a DFXML file (for example, to provide summaries for each file extracted from a disk image). Figure 3 shows the use of DCMI to annotate a disk image with a publisher, abstract, acquisition date, and sector size.

4. An Object-Oriented API for Forensic Processing

This section presents two Python modules that make it easy to write small programs that can perform complex forensic processing.

4.1. The dfxml.py and fiwalk.py Python modules

`dfxml.py` is a Python module that reads DFXML files and creates Python objects that directly map to DFXML's `<volume>`, `<fileobject>` and `<byte_run>` structures. Each object is then presented to a callback function for further processing.

Python provides two radically different models and corresponding interfaces for processing XML streams. The preferred approach is to use Python's SAX (Simple API for XML) parser. This second approach is generally faster and uses

```
<creator version="1.0">
  <program>fiwalk</program>
  <version>0.6.8</version>
  <build_environment>
    <compiler>GCC 4.2</compiler>
    <compilation_date>2011-03-17T18:47:41</compilation_date>
    <library name="tsk" version="3.2.0"></library>
    <library name="afflib" version="3.6.6"></library>
    <library name="libewf" version="20101215"></library>
  </build_environment>
  <execution_environment>
    <os_sysname>Darwin</os_sysname>
    <os_release>10.7.0</os_release>
    <os_version>Darwin Kernel Version 10.7.0: Sat Jan 29 15:17:16
    PST 2011; root:xnu-1504.9.37~1/RELEASE_I386</os_version>
    <host>demo.example.com</host>
    <arch>i386</arch>
    <command_line>fiwalk -x /dev/null</command_line>
    <uid>501</uid>
    <username>simsong</username>
    <start_date>2011-04-03T15:35:56-0600</start_date>
  </execution_environment>
</creator>
```

Figure 8: The `creator` element contains information about the program that was used to create the DFXML.

a smaller amount of memory, but is difficult for many programmers to master because it requires the creation of callback functions invoked for each tag or section of parsed character data that the parser encounters. The `dfxml.py` module provides these callbacks and processes the tags, providing the programmer with a simplified, higher-level API.

The design of the Python module means that constant memory is required for forensic tools whose primary mode of operation is to select files, process them, and then proceed to the next file. But the overhead of `dfxml.py`'s fileobjects is so small (typically between 100 and 1000 bytes per `<fileobject>`), that all the fileobjects for a file system with even millions of files can be processed in memory on a 32-bit system. This is useful when performing timeline analysis and correlations.

It is frequently convenient to have programs process disk images directly, without the need to first produce a DFXML file. The Python module `fiwalk.py` will run the *fiwalk* program and pipe the results into the `dfxml.py` module. Currently the XML file is not cached on the hard drive, although such caching could be added.

Sometimes it is advantageous to transform XML and produce an output file. DFXML has two approaches. An easy but inefficient way to do this using the framework is to forgo the SAX-based interface and instead use a second API within `dfxml.py` that relies on Python's `xml.dom.minidom` class. This class, based on the DOM (Hors, Hégaret, Wood, Nicol, Robie, Champion, and Byrne, 2004), allows read-write access to the XML.

Internally the `fileobject` object returned by the SAX-based functions belongs to a subclass called `fileobject_sax` while the `fileobject` returned by

Method	Description
filename()	Name of the file
filesize()	Size of the file in bytes
ext()	Returns the file extension as a lowercase string
ctime()	Metadata change time
atime()	File access time
crtime()	File creation time
mtime()	File modify time
alocated()	True if file is allocated
file_present()	True if the file is "present" in disk image
has_contents()	True if the file has one or more bytes on disk
byte_runs()	Returns an array of byte_run objects.
contents()	Byte array of file's contents
tempfile()	Returns a named temporary file with file's contents. Optionally calculates MD5 and SHA1 of the file as it is written to the disk.
toxml()	Returns an XML block associated with the fileobject

Table 1: Methods supported by the fileobject class.

the DOM-based functions belongs to the `fileobject_dom` subclass. Both subclasses have the same `fileobject` super-class; the class structure hides this implementation detail and allows either (or both) approaches to be used for processing forensic images. It is also possible to use the SAX API to ingest, process and emit modified DFXML, as the *dfxml.py* module includes support for XML generation. More work is needed in this area for an easy-to-use solution, however.

4.2. The fileobject object

Fileobjects support a straightforward API (see Table 1) in which most of the quantities of forensic interest can be retrieved with a function call.

4.3. Using fileobjects

It is relatively simple to obtain and work with the fileobjects associated with a disk image. For example, the program shown in Figure 9 will print the parti-

```
import fiwalk
def proc(fi):
    print(fi.partition(),fi.filename(),fi.filesize())
f = open("small.dmg","r")
fiwalk.fiwalk_using_sax(imagefile=f,callback=proc)
```

Figure 9: Accessing fileobjects using SAX with the callback interface.

tion number, filename and filesize of all the files contained within a disk image `small.dmg`.

Python's built-in functions for list processing make it relatively easy to operate on collections of fileobjects. For example, if `fobjs` is the list of fileobjects that match a certain criteria, Python's built-in `filter()` function can be used to select all of the fileobjects that have a length between 16 and 32 bytes, inclusively:

```
myfiles = filter(lambda x: 16 <= x.filesize() <=32, fobjs)
```

Figure 10 shows a more sophisticated program that reads all of the files in a disk image and produces a sorted timeline.

4.4. Accessing file contents

Fileobjects can also be used to access the content of the files that they point to. The primary way to access a file's contents are through the `contents()` method, which returns a string of the file's contents, and the `tempfile()` method, which copies the contents of the file out of the image and places it in a temporary file in the host file system, optionally calculating the MD5 and/or SHA1 in the process. By default both of these methods access the disk image provided when the objects were created, but both can also be used to access data from another image specified as an optional argument. This can be useful to see whether individual

```
import fiwalk, dfxml, sys
timeline = []

def process(fi):
    timeline.append((fi.mtime(),fi.filename()," modified"))
    timeline.append((fi.atime(),fi.filename()," accessed"))
    timeline.append((fi.ctime(),fi.filename()," metadata changed"))

fiwalk.fiwalk_using_sax(imagefile=open(sys.argv[1]),callback=process)
timeline.sort()
for record in timeline:
    print("\n".join(record))
```

Figure 10: A small Python program using *fiwalk.py* and *dfxml.py* that prints a timeline of a disk image.

files have changed between images (the `file_present()` method implements this functionality by checking to see if the hash value of the file has changed).

4.5. Helper classes

The *dfxml.py* module also contains a few helper classes that aid in processing DFXML files.

The *byte_run* class represents byte runs. This class can perform basic set-of-sector operations such as determining the intersection of two *byte_runs*, determining if a sector from the drive is within a byte run, and producing XML associated with a run.

A *dftime* class represents the ISO 8601 times found in DFXML files. It can also operate with epoch-based times that may be found in some XML files or other data structures. This class can also interconvert between ISO 8601 and the two other time standards available to Python programs.

5. DFXML Tools

This section presents several tools that emit and consume DFXML.

5.1. fiwalk

fiwalk is a tool that ingests disk images and emits DFXML objects corresponding to all allocated, deleted, and orphan files in any file systems found on the disk.

fiwalk is designed to automate the initial forensic analysis of a disk image and in so doing eliminate many of the points of confusion typically exhibited by those who are not intimately familiar with file system forensics. Specifically:

- *fiwalk* can be applied to live file systems, raw devices, or disk images.
- As *fiwalk* is based on SleuthKit, the program can operate on disk images in any format that SleuthKit supports.
- If the target contains a single file system, *fiwalk* automatically processes all of the files and inodes in the file system. If the target is partitioned, *fiwalk* automatically processes all of the partitions. SleuthKit beginners are frequently confused as to whether or not they should provide a `-o 63` option with the filesystem-level commands. *fiwalk* removes this point of confusion.

When creating XML files from disk image files in AFF or EnCase format, *fiwalk* will extract metadata such as the serial number of the imaged disk or the experimenter's notes, and include this information in the resulting XML file.

fiwalk features a plug-in architecture that can automatically run metadata extractors when files of specific types are encountered. For example, the JPEG metadata extractor can automatically extract EXIF information when JPEGs are encountered. XML namespaces are used to prevent conflict between tags. The results of the metadata extractors are automatically incorporated into the output streams.

Three plug-in interfaces have been designed for *fiwalk*:

dgi Similar to the Apache web server *CGI* interface, the extractor runs as a standalone process with the file specified on the command line. Extracted metadata are provided back to *fiwalk* on the STDOUT as a set of *name:value* pairs. *fiwalk* automatically collects these pairs, escapes them as necessary, and turns them into the appropriate XML.

shlib *fiwalk* loads a shared library into its address space using the same API that was developed for the *bulk_extractor* forensics tool.

jvm *fiwalk* communicates with a metadata extractor written in Java using Java's Invocation API.

The publicly released version of *fiwalk* supports only the *dgi* interface. Several plug-ins are distributed with the program:

docx_extractor.py extracts document properties from the Microsoft Office Open XML file.

ficlam.sh uses the open source Clam AV anti-virus system to scan files for malware.

jpeg_extract uses *libexif* to extract EXIF information from JPEG files.

odf_extractor.py extracts document properties from files in the Open Office format.

word_extract.java extracts document properties from legacy Microsoft Office Compound Document files (DOC, XLS and PPT) using the *wv*Lachowicz and McNamara (2006) system.

An example of extracted metadata appears in Figure 11.

```
<msword xmlns:msword="http://afflib.org/fiwalk/msword">
  <msword:msole-codepage>1255</msword:msole-codepage>
  <msword:Generator> Microsoft Word 9.0 </msword:Generator>
  <msword:Creator> EPOCH </msword:Creator>
  <msword:Revision> 1 </msword:Revision>
  <msword:Number-of-Pages>1</msword:Number-of-Pages>
  <msword:Number-of-Words>0</msword:Number-of-Words>
  <msword:Created>2003-05-10T12:12:00Z</msword:Created>
  ...
</msword>
```

Figure 11: An excerpt of the metadata extracted from a Microsoft Word file that accompanies a Grand Theft Auto Mission Pack, generated using *fiwalk* and the Microsoft Office Compound Document metadata extractor.

5.2. idifference.py

Examiners are frequently interested in understanding the differences between two DFXML files. An obvious case is when a hard drive is imaged, used, and then imaged again—for example, before and after an application is installed, to determine the application footprint.

idifference.py is a Python program that compares two DFXML files and reports the differences on the fileobjects that they contain. The changes currently detected and reported include:

- Files deleted
- Files created
- Files moved or renamed (determined because a file was created and another deleted that have the same cryptographic hash)
- Files that were modified without a change to the modification timestamp (indicative of a hardware problem, software error, or attempted malicious activity)

- Files that have had their modification timestamps changed without a corresponding change to file contents.

Currently *idifference.py* produces its output as a human-readable file. In the future it will also produce a DFXML file so that the difference processing can in turn be ingested by other tools.

5.3. imicrosoft_redact.py

Computer forensics researchers need to distribute disk images of computer systems to allow for the duplication of results and the validation of forensic tools (Garfinkel et al., 2009). Such distribution can be problematic, as a disk image of a computer running Microsoft Windows can be readily turned into a virtual machine and boot, potentially violating Microsoft's copyright on the files contained therein. However, such uses may be permissible under US copyright law under the *fair use* exemption, provided that the use is for "teaching, scholarship [or] research," and provided that a competent court concludes the use is fair. Under Section 107 of the Copyright Act, courts consider four factors in making their determination:

1. The purpose and character of the use, including whether such use is of commercial nature or is for nonprofit educational purposes
2. The nature of the copyrighted work
3. The amount and substantiality of the portion used in relation to the copyrighted work as a whole
4. The effect of the use upon the potential market for, or value of, the copyrighted work (U.S. Copyright Office, 2009).

To this end, the DFXML distribution includes a tool that can modify executables contained within a disk image so that the image cannot be turned into a workable virtual machine. The tool, *imicrosoft_redact.py*, further notes what files have been modified, and records the cryptographic hash of the files before and after modification. This allows individuals with copies of these files (for example, if they subscribe to the Microsoft Developer Network) to restore the corrupted files.

This approach allows disk images of Microsoft Windows installations to be distributed under the fair use doctrine for the purpose of digital forensics research because:

1. The purpose of the distribution is for research and nonprofit educational use.
2. The information that is distributed is a non-working derivative work of Microsoft Windows.
3. The value of Microsoft Windows is not impacted by the distribution of the derivative work.

6. Conclusion

This article describes Digital Forensics XML (DFXML), an XML language for digital forensics research and interchange. DFXML is designed to be an interchange format between forensic tools. The abstractions represented in DFXML have been specifically chosen to represent digital forensic processing steps, allowing for ease of generating and ingesting DFXML objects.

6.1. Future work

The expressive power of DFXML can be used for many purposes other than documenting the results of a forensic investigation. For example:

Application and malware profiles DFXML can be used to describe the collection of files that make up an application, the Windows Registry or Macintosh *plist* information associated with an application, document signatures, and network traffic signatures. Using DFXML it should be possible to distribute a machine-readable application profile that will allow a tool to automatically determine if an application is present on a hard drive, when it was last used, or if an application was used and later uninstalled. This use is very similar to a primary use case for MITRE's MAEC project.

Targeting It would be useful to expand DFXML to include identity information associated with the targets of investigations. For example, there needs to be a canonical representation for GPS coordinates, email addresses, credit card numbers, phone numbers, and so on. Such representations will make it dramatically easier for practitioners to exchange target lists, watch lists, stop lists, and the like.

User profiles DFXML can describe the tasks that a user engages in, which applications the user runs, when they run, and for what purpose. Using DFXML it should be possible to create profiles indicative of specific users. Alternatively it should be possible to programmatically extract information pertaining to a user and provide this to an automated reporting tool.

Internet footprint DFXML can document both the information that a user contributes to the global Internet and the information required to access it (**?**). It should be possible to create a tool using DFXML that finds Internet residue on a hard drive and uses that information to prepare an evidence-based briefing.

The approach presented here for using Python to automate forensic processing can be easily extended to existing all-in-one forensic systems such as EnCase, FTK and PyFlag. It would certainly be advantageous to the forensic community if a single simple but powerful programming environment could run within all these applications. One of the advantages of the object-oriented system described here is that it can easily be applied to parallel computing environments.

6.2. Availability

The *fiwalk* program, *dfxml.py* and *fiwalk.py* modules, and all of the applications discussed in this article can be downloaded from `http://www.afflib.org` as part of the *fiwalk* distribution. The software is in the public domain and can be used by anyone for any purpose.

6.3. Acknowledgments

George Dinolt, Christophe Grenier, Joshua Gross, Jesse Kornblum, Neal Krawetz, Alex Nelson, Adam Russell, Elisabeth Rosenberg, Tony Zuccaro and the anonymous reviewers all provided useful feedback and criticism regarding the design of DFXML. Portions of this work were funded by NSF Award DUE-0919593.

The views and opinions expressed in this document represent those of the author and do not necessarily reflect those of the US Government or the Department of Defense.

References

Alink, W., Bhoedjang, R., Boncz, P., de Vries, A., 2006a. Xiraf xml-based indexing and querying for digital forensics. Digital Investigation 3S, S50–S58. `http://www.dfrws.org/2006/proceedings/7-Alink.pdf`

Alink, W., Jijkoun, V., Ahn, D., de Rijke, M., Boncz, P., de Vries, A., 2006b. Representing and querying multi-dimensional markup for question answering. In: Proceedings of the 5th Workshop on NLP and XML: Multi-Dimensional

Markup in Natural Language Processing. NLPXML '06. Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 3–9.
http://portal.acm.org/citation.cfm?id=1621034.1621036

Allen, B., Dec. 2009.
http://sourceforge.net/projects/libewf/files/libewf-java/

Biron, P. V., Malhotra, A., Oct. 28 2004. XML schema part 2: Datetypes.
http://www.w3.org/TR/xmlschema-2/#isoformats

Cameron, R. D., Herdy, K. S., Lin, D., 2008. High performance xml parsing using parallel bit stream technology. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds. CASCON '08. ACM, New York, NY, USA, pp. 17:222–17:235.
http://doi.acm.org/10.1145/1463788.1463811

Carrier, B., Oct. 28 2010. Sleuthkit 3.2.0.
http://www.sleuthkit.org/sleuthkit/

Cohen, M. I., Garfinkel, S., Schatz, B., 2009. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. In: Proceedings of DFRWS 2009. Elsevier, Montreal, Canada.

Common Digital Evidence Storage Format Working Group, 2007. DFRWS CDESF working group.
http://www.dfrws.org/CDESF/index.shtml

Diaz, E., Sep. 20 2005. Exploiting MD5 collisions (in c#).
http://www.codeproject.com/KB/security/HackingMd5.aspx

Dima, A., 2006. WiReD—windows registry dataset—BETA release CD ISO. National Institute of Standards and Technology.
`http://www.nsrl.nist.gov/Downloads.htm`

Dublin Core Metadata Initiative, Oct. 11 2010. Dublin core metadata element set, version 1.1.
`http://www.dublincore.org/documents/dces/`

Farmer, D., Venema, W., 2005. Forensic Discovery. Addison-Wesley Professional, New York, NY.

Frazier, M., Jan. 26 2010. Combat the apt by sharing indicators of compromise. M-unition.
`https://blog.mandiant.com/archives/766`

Garfinkel, S., Feb. 2006. AFF: A new format for storing hard drive images. Communications of the ACM.

Garfinkel, S., Parker-Wood, A., Huynh, D., Migletz, J., Dec. 2010. A solution to the multi-user carved data ascription problem. IEEE Transactions on Information Forensics and Security 5, 868–882.

Garfinkel, S. L., Farrell, P., Roussev, V., Dinolt, G., Aug. 2009. Bringing science to digital forensics with standardized forensic corpora. In: Proceedings of the 9th Annual Digital Forensic Research Workshop (DFRWS). Elsevier, Quebec, CA.

Google, 2011. Protocol buffers.
`http://code.google.com/apis/protocolbuffers/`

Grenier, C., 2011. Photorec.
`http://www.cgsecurity.org/wiki/PhotoRec`

Guidance Software, 2007. EnScript Programs Version 6.3 User Manual. Guidance Software, Inc., Pasadena, CA.

Hack, M., Meng, X., Froehlich, S., Zhang, L., 27 2010-oct. 1 2010. Leap second support in computers. In: Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on. pp. 91 –96.

Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S., Apr. 2004. Document object model (dom) level 3 core specification.
`http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`

Howell, C., 2009. Regripper.
`http://regripper.wordpress.com/`

Huynh, D., 2008. Exploring and validating data mining algorithms for use in data ascription. Master's thesis, Naval Postgraduate School, Monterey, CA.
`http://theses.nps.navy.mil/08Jun_huynh.pdf`

IEEE, 2004. The open group base specifications issue 6, ieee std 1003.1, 2004 edition.
`http://pubs.opengroup.org/onlinepubs/009604599/xrat/xbd_chap04.html`

ISO, 2000. ISO 8601:2000. Data elements and interchange formats — Information interchange — Representation of dates and times. International Standards

organization, Geneva, Switzerland.
`http://www.iso.ch/cate/d26780.html`

Jones, R. W. M., 2009. hivexml—convert windows registry binary 'hive' into xml. Red Hat Inc.
`http://libguestfs.org/`

Kloet, B., Metz, J., Mora, R.-J., Loveall, D., Schreiber, D., 2008. libewf: Project info.
`http://www.uitwisselplatform.nl/projects/libewf/`

Kornblum, J., Jun. 26 2011. md5deep and hashdeep–latest version 3.9.2.
`http://md5deep.sourceforget.net`

Lachowicz, D., McNamara, C., 2006. wvware.
`http://wvware.sourceforge.net`

Levine, B. N., Liberatore, M., 2009. Digital Investigation 6, S48–S56.
`http://www.dfrws.org/2009/proceedings/p48-levine.pdf`

Meijer, R., 2011. The carve path zero-storage library and filesystem.
`http://ocfa.sourceforge.net/libcarvpath/`

Microsoft, Dec. 30 2008. Microsoft security advisory (961509) research proves feasibility of collision attacks against MD5.
`http://www.microsoft.com/technet/security/advisory/961509.mspx`

Migletz, J., 2008. Automated metadata extraction. Master's thesis, Naval Post-

graduate School, Monterey, CA.
`http://theses.nps.navy.mil/08Jun_Migletz.pdf`

Morgan, T. D., Jun. 9 2009. The windows nt registry file format version 0.4.
`http://sentinelchicken.com/data/TheWindowsNTRegistryFileFormat.pdf`

Python Software Foundation, 2010. xml.sax: Support for sax2 parsers. Python v2.7.1 documentation.
`http://docs.python.org/library/xml.sax.html`

Rodriguez, S., Jan. 21 2003. Import/export registry sections as XML. The Code Project.
`http://www.codeproject.com/KB/system/registryasxml.aspx`

Selinger, P., Jan. 17 2009. MD5 collision demo.
`http://www.mscs.dal.ca/~selinger/md5collision/`

Shayne, E., Aug. 2001. Regxml.
`http://www.eshayne.com/RegXML/`

Socha, G., 2011. The electronic discovery reference model XML.
`http://edrm.net/projects/xml`

Tang, Z., Ding, H., Xu, M., Xu, J., 2009. Carving the windows registry files based on the internal structure. In: Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering. ICISE '09. IEEE Computer Society, Washington, DC, USA, pp. 4788–4791.
`http://dx.doi.org/10.1109/ICISE.2009.379`

Thomassen, J., Apr. 11 2008. Forensic analysis of unallocated space in windows registry hive files. Master's thesis, University of Liverpool.

Turner, P., Aug. 2005. Unification of digital evidence from disparate sources (digital evidence bags). In: Proceedings of the 2005 Digital Forensics Research Workshop. Elsevier, London, England.

U.S. Copyright Office, 2009. Fair use.
`http://www.copyright.gov/fls/fl102.html`

US Department of Justice, US Department of Homeland Security, 2011. Terrorist watchlist person data exchange standard overview.
`http://www.niem.gov/TWPDES.php`

Zhang, W., van Engelen, R. A., 2006. Tdx: a high-performance table-driven xml parser. In: Proceedings of the 44th annual Southeast regional conference. ACM-SE 44. ACM, New York, NY, USA, pp. 726–731.
`http://doi.acm.org/10.1145/1185448.1185606`

Zyp, K., Court, G., Nov. 22 2010. A json media type for describing the structure and meaning of json documents.
`http://tools.ietf.org/html/draft-zyp-json-schema-03`