Automating Disk Forensic Processing with SleuthKit, XML and Python

Simson L. Garfinkel

Abstract

We have developed a program called `fiwalk` which produces detailed XML describing all of the partitions and files on a hard drive or disk image, as well as any extractable metadata from the document files themselves. We show how it is relatively simple to create automated disk forensic applications using a Python module we have written that reads `fiwalk`'s XML files. Finally, we present three applications using this system: a program to generate maps of disk images; an image redaction program; and a data transfer kiosk which uses forensic tools to allow the migration of data from portable storage devices without risk of infection from hostile software that the portable device may contain.

Keywords: Computer Forensics; XML; Sleuth Kit; Python

I. Introduction

In recent years we have found many potential applications for computer forensic tools that extend beyond the traditional domain of law enforcement and e-discovery. Many researchers that we have spoken with would like to be able to work with data resident on hard drives or USB memory sticks without having to learn the specifics of disk image formats, partitions, file systems, and so forth. Others would like to develop turnkey applications that can process disk images without user intervention.

Faced with such needs, the standard approach is to write programs that run SleuthKit[2] commands as subprocesses and process the results. But SleuthKit's interface is not well suited to this task. The fields output using the "machine" option from commands such as `fls` are not labeled and changed significantly between version 2.0 and 3.0. Other commands, such as `mls`, must have their output manually parsed or "screen scraped." Putting together working applications requires considerable forensic knowledge, and the resulting applications are not efficient because they require multiple invocations of the SleuthKit programs.

To assist users who wanted to create programs that can automatically process disk images, we have developed a methodology for automating the processing of forensic data. This methodology uses a new program that we have developed to process a disk image into a single XML structure that represents all of the file system and document metadata resident within a disk image. The XML structure can then be used in conjunction with the disk image and a Python class library to allow a wide variety of automated forensic processing using simple scripting.

The remainder of this introduction discusses other approaches to automated disk forensic processing. Section II presents `fiwalk`, the program that we have developed for automatically extracting file system and document metadata from disk images and transforming that information into XML. Section III presents `fiwalk.py`, a Python module that we have written to ease the automated processing of the XML that `fiwalk` produces. Section IV presents several applications that we have written on this platform.

A. Contributions of this Research

This paper presents a new approach for creating special-purpose forensic tools. Based on SleuthKit, XML and the Python programming language, this approach makes it relatively easy for programmers to create tools that can perform forensic processing without the need to master domain-specific knowledge.

B. Prior Work

Despite the growing interest in computer forensics, there has been little work on approaches for automated forensic analysis. This is especially surprising given the obvious importance of automation for both intelligence and law enforcement.

Forensic analysis today is a largely manual process performed using software such as EnCase[8] and FTK[1]. EnCase does come with support for its own object-oriented "EnScript" scripting language for

automating tasks. The language is similar to Java and includes classes that support reading files, file system structures, and metadata, and for storing the results in "folders" and "bookmarks" that are part of the EnCase user interface. But EnScript is not used by any other program, and it is relatively difficult to get significant technical information on the language. (Guidance Sofware's EnCase v6 EnScript programming cost has a tuition of $3,295.00 per student.)

PyFlag is an open source forensic system developed by the Australian Federal Police. PyFlag can be scripted using its own pyflash scripting language for simple tasks and Python for complex ones[5]. Unfortunately, PyFlag is not widely used because of its complexity and difficulty of deployment.

II. Producing XML with `fiwalk`

After developing a large collection of disk images and teaching several day-long training courses in disk forensics, we decided to develop a new forensic extraction utility based on Carrier's SleuthKit Library[2]. Our tool is called `fiwalk`, short for "file and inode walk."

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

- `fiwalk` can be applied to live file systems, raw devices, or disk images.
- Like the SleuthKit executables, `fiwalk` recognizes disk images in any format.
- 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. In our experience with 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.

`fiwalk` uses Sleuth Kit's `tsk_vs_part_walk()` to walk the image partitions, `tsk_fs_dir_walk()` to walk all directories, and `tsk_fs_file_walk()` to extract the individual data blocks for each file that TSK can recognize. The results from running the SleuthKit libraries can be output in three different formats:

1) As a human-readable "walk file" which is useful for debugging.
2) As an Attribute Relationship File Format (ARFF) suitable for use with the Weka datamining toolkit.
3) As an XML file, with embedded `<partition>` elements for each partition and `<fileinfo>` elements for each file.

When processing XML files from disk image files in AFF or EnCase format, `fiwalk` will also 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, creating tags in the same format as the `afxml` program. These tags make it easy for programs that consume the XML file to make use of this image-specific metadata.

A. *`fiwalk`'s XML*

The XML block that `fiwalk` produces has four main parts:

1) Information about the specific tools and data that were used to create the XML file (*e.g.*, the version number of fiwalk, TSK and AFF; the disk image filename and associated disk image metadata). This information allows us to automatically re-generate the XML files as needed when new features are added or bugs are fixed in the underlying tool set. Including this information is important for establishing the provenance of the resulting data.
2) Information about the disk image itself (*e.g.*, the sector size and number sectors). This is useful for producing overall statistics about the corpus.
3) Information about the partitions that were identified (*e.g.*, the partition offset from the start of the device, the file system type, etc.)
4) Information about each file (*e.g.*, the file name, file size, etc). Most forensic analysis relies on this information.

The per-file information is stored within `<fileobject>...</fileobject>` tags. Figure 1 shows an example of such an XML block. All of the fields from SleuthKit `TSK_FS_META` structure are broken out into their own XML tags.

The per-file XML block also contains information about the contents of the file, including the file's MD5 and SHA1 hash values and result of running the file contents through `libmagic`[6]. Finally, the per-file XML block contains a map of where the file's contents are stored on the physical disk.

```
<fileobject>
  <id>10</id>
  <filesize>16384</filesize>
  <partition>1</partition>
  <ALLOC>1</ALLOC>
  <USED>1</USED>
  <mtime>1227139058</mtime>
  <atime>1227081600</atime>
  <crtime>1227139058</crtime>
  <filename>file3</filename>
  <libmagic>ASCII text</libmagic>
  <byte_runs>
    <run fs_offset='31744' img_offset='64000' file_offset='0'
         len='2048'/>
    <run fs_offset='35840' img_offset='68096' file_offset='2048'
         len='14336'/>
  </byte_runs>
  <md5>3b4af47f8b542fb5d1bdaeec34563d89</md5>
  <sha1>b614d783fee50f053bd99da89401b89b013487a4</sha1>
</fileobject>
```

Fig. 1. An example of a fileobject XML tag generated by fiwalk, with a few extra spaces added for readability.

The XML file map is represented as an array of extents. Each extent consists of a physical position, a logical location within the file, a length, and a mode. Starting position is measured from the beginning of the disk image, a decision made based on the difficulty that we and our students have had using SleuthKit's user-level commands, many of which require that the user know if the file system is on a partitioned disk or not. All measurements are made in bytes, which eliminates the need to keep track of the physical drive's sector size. (The decision to use bytes instead of sectors also makes working with CDROMs and new 4096-byte sector drives easier, since it eliminates the need to track of which sector size is currently applied.) Mode can be *raw*, *zero filled* (in which the start position is ignored and the extent is assumed to contain zeros), and *compressed* (in which case the run needs to be first decompressed with the appropriate decompression algorithm).

`fiwalk`'s XML has evolved for programmer ease-of-use, rather than for the purpose of optimizing compactness of representation. As a result there exist fields that are redundant. But these redundancies significantly simplify processing when XML subtrees are extracted and used as documents of their own.

B. Remote File Discovery and Extraction Using XML

We have used the XML produced by `fiwalk` to allow specific files from disk images stored at one location to be browsed and extracted over the Internet from another location. This has facilitated collaboration on a large-scale forensic data collection project between multiple locations without necessitating the movement of entire disk images between locations.

In our project, approximately 2000 physical disk drives were obtained and imaged. The resulting archive of disk images required 4TB of disk space (compressed). One of our research partners was located in a physically distinct location.

Although we could have sent our research partner 8 500GB hard drives containing the 4TB of disk images, we decided against this approach because of the excessive handling that would have been required, and because our research partner did not have the forensic tools for extracting files from disk images. Instead we used `fiwalk` to create an XML file for each disk, and placed all of the XML files in a password-protected directory on our SSL-enabled web server. Our research partner downloaded all of the XML files and scanned each one for files matching a specific criteria. For each matching file the partner issued an XML-RPC call to our secure server, providing data from the `byte_runs` element. The XML-RPC server simply opened the corresponding disk image file, performed a `seek` and `read` for each byte run, concatenated the results, and sent them back to the collaborator. Files could be verified by comparing the hash of the downloaded file with the hash present in the XML.

```
<Editing-Duration>2009-04-22T19:24:48Z</Editing-Duration>
<msole-codepage>1255</msole-codepage>
<Generator> Microsoft Word 9.0 </Generator>
<Last-Modified>2003-05-10T12:12:00Z</Last-Modified>
<Creator> EPOCH </Creator>
<Revision> 1 </Revision>
<Number-of-Pages>1</Number-of-Pages>
<Number-of-Words>0</Number-of-Words>
<Title> Grand Theft Auto: London 1969 </Title>
<Created>2003-05-10T12:12:00Z</Created>
<Subject>  </Subject>
<Template> Normal </Template>
<Keywords>  </Keywords>
<Description>  </Description>
<Number-of-Characters>0</Number-of-Characters>
<Security-Level>0</Security-Level>
<Last-Saved-by> EPOCH </Last-Saved-by>
<msole-codepage>1255</msole-codepage>
<Number-of-Lines>1</Number-of-Lines>
<Document-Parts>[(0,  Grand Theft Auto: London 1969 )]</Document-Parts>
<Number-of-Paragraphs>1</Number-of-Paragraphs>
<Unknown1>0</Unknown1>
<Company>  </Company>
<Scale>FALSE</Scale>
<Links-Dirty>FALSE</Links-Dirty>
<Unknown3>FALSE</Unknown3>
<Document-Pairs>[(0,\327\251\327\235), (1,1946157057)]</Document-Pairs>
<Unknown6>FALSE</Unknown6>
<Unknown7>592544</Unknown7>
```

Fig. 2. A section of the metadata extracted from a Microsoft Word file that accompanies a Grand Theft Auto Mission Pack.

Using this so-called *remote exploitation methodology* our research partner was able to scan our corpus of disk images, locate desirable information, and download more than 600,000 individual files—all with minimal human intervention (once the software was written, of course).

C. Metadata plug-ins

`fiwalk` also features a plug-in architecture which 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. The results of the metadata extractors are automatically incorporated into the output streams.

The `fiwalk` plug-in system supports two plug-in interfaces. The "dgi" interface is similar to the Apache web server "cgi" interface: the extractor is run as a stand-alone process with the file specified on the command line; extracted metadata is provided back to `fiwalk` on the standard out as a set of "name:value" pairs. `fiwalk` automatically collects these pairs, quotes them, and turns them into the appropriate XML.

The second interface that we have designed runs the metadata extractor in a JVM subprocess. Communication with the subprocess is over a TCP connection. This approach avoids having to start up a new process for each file processed and also allows developers to write metadata extractors in Java. We have designed this interface and tested it using a dgi-to-JVM adapter, but have not implemented the full TCP-based communication system due to lack of interest on the part of our students.

We have developed four plug-ins: one that uses `libexif` to extract EXIF information; one that uses `wv2` to extract metadata from Microsoft Office Compound Document files (DOC, XLS and PPT); one that extracts metadata from Microsoft's new Open Office XML format; and one that extracts metadata from the Open Document Format. An example of extracted metaata appears in Figure 2.

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
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 file object

TABLE I
METHODS SUPPORTED BY THE FILEOBJECT CLASS.

III. WORKING WITH FIWALK'S XML FROM PYTHON

We have also created `fiwalk.py`, a Python module which makes it easy to work with the XML files produced by `fiwalk`. As has been previously noted[5], Python makes an excellent language for writing forensic tools because of flexible object model, its built-in garbage collection, and its interactive prototyping environment. We further find that Python's ease of working with large lists and its facility of handling XML to be essential for building batch forensic applications.

The `fiwalk.py` module reads the `fiwalk` XML structure and produces python object for each file specified in the XML. Individual properties of the files can be accessed using Python access methods described in Table I.

Python provides two radically different models and corresponding interfaces for processing XML streams. The first is the `xml.dom.minidom` class, based on the World Wide Web Consortium's Document Object Model[9]. Python's DOM implementation reads the entire XML stream into memory and allows read/write access to the XML document. The second is based on the SAX (Simple API for XML) and is implemented with Expat. This second approach is generally faster and uses less memory than the first, but it does not allow read-write access. These two approaches are complementary, the first providing more flexibility, the second providing speed and being able to work with significantly larger documents.

Because neither the DOM nor the SAX approach is better, `fiwalk.py` provides both a DOM-based and an SAX-based interface for processing the `fiwalk` XML files:

- `fiwalk_using_sax()` This interface uses the expat-based parser. The caller can provide either an XML file or else an imagefile, in which case `fiwalk` is automatically run in a child process. The caller must provide a callback function which is called for each `fileobject` that is created.
- `fileobjects_using_dom()` This interface uses the DOM-based parser. As before, the caller can provide either an XML file or an imagefile. The function returns a python minidom object for the entire XML structure as well as a list of all the `fileobject` elements extracted into a single array.
- `fileobjects_using_sax()` This interface uses the expat-based parser, but returns an array of all the `fileobject` objects. It is provided for completeness.

Internally the `fileobject` object returned by the DOM and SAX methods are somewhat different. The DOM-based functions return objects that belong to a `fileobject` subclass called `fileobject_dom`, while the SAX-based functions return objects that belong to a subclass called `fileobject_sax`. The `fileobject` super-class hides this implementation detail and allows either (or both) approaches for processing forensic images.

```
import fiwalk
f = open("small.dmg")
(doc,fobjs) = fiwalk.fileobjects_using_dom(imagefile=f)
for fi in fobjs:
    print(fi.partition(),fi.filename(),fi.filesize())
```

Fig. 3. Accessing file objects using the DOM interface.

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

Fig. 4. Accessing file objects using SAX with the callback interface.

A. Getting file objects from images

It is relatively simple to obtain and work with the file objects associated with a disk image. For example, the program shown in Figure 3 will print the partition number, filename and filesize of all the files contained within a disk image `small.dmg` using the DOM interface:

The `fileobjects_using_dom()` function actually takes three arguments:

imagefile	An open file corresponding to a disk image.
xmlfile	An open file corresponding to the XML file for the image.
flags	An integer flag that controls the behavior of `fiwalk`.

As noted above, `fiwalk` runs as a child process if the `xmlfile` is not provided. If `imagefile` is provided the individual file objects will be able to access the content of each file; otherwise the objects can only be used to manipulate metadata.

The program shown in Figure 4 does the same thing using SAX and the callback interface. This interface is significantly faster and uses less memory because only one file object exists at a time.

The `fileobjects_using_sax` interface provides a middle ground between the flexibility of the DOM interface and the speed of the callback interface: all of the file objects exist at the same time, but they are based on dictionaries and are not XML documents (Figure 5).

B. Working with file objects

The fileinfo objects are full-fledged Python objects. The program in Figure 6 uses the callback mechanism to calculate the mean and standard deviation of files of a given type in a disk image. The callback is a user-supplied function with a single fileinfo object as the argument. (The code would be simpler and would not need the `global` statement of Python had proper closures.)

Python's built-in functions for list processing makes it relatively easy to operate on collections of file objects. For example, if `fobjs` is the list of all the file objects from the first example, we can use the built-in function `filter()` to select all of the file objects that have a length of 15 bytes:

```
myfiles = filter(lambda x:x.filesize()==15, fobjs)
```

For those who find Python's `filter` statement difficult to comprehend, the code can be rewritten as:

```
myfiles = []
for fi in fobjs:
    if x.filesize()==15: myfiles.append(fi)
```

This code fragment uses Python's list comprehensions to create an array of the sizes of all the files on in the disk image: It then draws a histogram of the values using the `matplotlib` package:

```
fobjs = fiwalk.fileobjects_using_sax(imagefile=open("small.dmg"))
sizes = [fi.filesize() for fi in fobjs]

from pylab import hist
hist(sizes,bins=100)
```

```
import fiwalk
f = open("small.dmg")
fobjs = fiwalk.fileobjects_using_sax(imagefile=f)
for fi in fobjs:
    print(fi.partition(),fi.filename(),fi.filesize())
```

Fig. 5. Using the SAX interface to access file objects as a list.

```
import fiwalk,math

sum_ = 0; sum_squares = 0; count = 0

def f(fi):
    global total,total2,count
    if fi.ext()=='txt':
        sum_ += fi.filesize()
        sum_squares += fi.filesize() ** 2
        count += 1

fiwalk.fiwalk_using_sax(imagefile=open("small.dmg"),callback=f)
print("count=%d  average=%g" % (count,sum_/count))
print("stddev=%g" % math.sqrt(sum_squares/count-(sum_/count)**2))
```

Fig. 6. A small python program demonstrating how to compute the average and standard deviation of all the files with the extension 'txt' in a disk image. Because this program uses the callback notation, it will handle a disk image of any size.

C. Accessing File Contents

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

D. Adding new methods to the file objects

Many Python programmers don't know that methods can be freely added to any class at runtime. This is useful for forensic analysis, as it allows new functionality to be embedded inside `fileobject` objects without subclassing the `fileobject` class (which can't be readily done without modifying the `fiwalk.py` file). To demonstrate with a trivial example, suppose for forensic analysis it was useful to have a boolean method which returned whether or not a file object was a dot file—that is, if the filename began with a period. This method could be defined and added to the fileobject base class as shown in Figure 7.

E. Annotating the DOM file object XML

Because file objects built using the DOM parser contain references to the file object's `<fileobject>` XML structure, the XML can be directly annotated using Python's `minidom` implementation. This annotation capability makes it relatively easy for even novice programmers to create forensic filters that can read the XML file associated with a disk image, perform an analysis on one or more files, annotate the XML for those files with the results of the analysis, and write out the results as a new XML file.

For example, if `fi` is a reference to a fileobject corresponding to a JPEG file, the very small Python function shown in Figure 8 will add a `<color>green</color>` tag to the file's XML.

```
def dotfile(self):
    import os.path
    return os.path.basename(self.filename()).startswith(".")
fiwalk.fileobject.dotfile = dotfile
```

Fig. 7. Python allows new methods to be added to existing classes at runtime.

```
def add_color_green_to_fidoc(fi):
    dom = fi.doc.ownerDocument
    color = dom.createElement("color")
    color.appendChild(dom.createTextNode("green"))
    fi.doc.appendChild(color)
```

Fig. 8. A Python function that would add a <color>green</color> XML tag to a file object's XML. This function can be expanded to perform additional processing and annotation; the resultant changes to the XML can then be saved.

IV. Three Simple Applications

In this section we will discuss three simple applications that we have created with this framework. What makes the applications "simple" is the fact that most of the complexity has been hidden in SleuthKit and the `fiwalk` module.

A. Ground Truth and Residual Data

A "map" of a disk image is an accurate account of the contents of each sector—or, conversely, a report listing all of the files contained within a disk image and the physical sectors where each file resides. Sector maps have been distributed with the solutions to the annual DFRWS carving challenges[3], [4]. Although such maps may not be useful when conducting after-the-fact forensic investigations, they are extremely useful for presenting the results of a forensic investigation to a third party, for evaluating the effectiveness of forensic tools, for for conducting research into the behavior of operating systems, and for forensic education.

The difficulty of creating a sector map is not to provide the sectors used by allocated files—`fiwalk` does this automatically—but to provide the sectors of files that are completely or mostly intact but for which the metadata has been overwritten. That is, the difficulty is to create a map of the data that can only be recovered with file carving or other advanced recovery techniques.

The authors of the DFRWS Carving Challenges solved the "map" problem by first constructing the map and then using it to build the disk image. This approach generates an accurate map but has the disadvantage that the resulting disk image is not a realistic representation of what an operating system might create.

Another approach for creating a map is to use self-identifying data—for example, each file might be filled with lines that say "this is line III of file JJJJ." This is the approach used by Garfinkel and Malan[7] and Stahlberg *et al,*[10]. This approach is relatively easy to implement as well and generates realistic disk images, but has the disadvantage of requiring purpose-built data for the analysis: it cannot be used with web cache data, for example.

Yet a third approach is to use an instrumented file system that creates a trace file which records the specific blocks where each file is written.

Although this approach may not be useful in an after-the-fact investigation, it is very useful in other situations—for example, monitoring a honeypot, or for preparing forensic test images that will be used in tool testing and education.

We have developed a new approach for building disk image maps that uses multiple disk images made from the same file system as it changes over time. File system metadata extracted from earlier disk images is used as guides for finding data stored in the final disk image. For example, consider a FAT32 file system used in a digital camera. The July 10th image of the file system might have a file `alice.jpg` split into 3 fragments. The file system is imaged. Later, on July 11th, the file `alice.jpg` is deleted and the directory entry for `alice.jpg` is used by a new file, `bob.jpg`. If sectors used by `alice.jpg` have not been overwritten, it is be possible to recover the `alice.jpg` image from the July 11th disk image by using information from the XML map of the July 10th image.

```
<fileobject>
  <id>10</id>
  <filesize>16384</filesize>
  <partition>1</partition>
  <ALLOC>1</ALLOC>
  <USED>1</USED>
  <mtime>1227139058</mtime>
  <atime>1227081600</atime>
  <crtime>1227139058</crtime>
  <filename>file3</filename>
  <libmagic>ASCII text</libmagic>
  <original>
    <image>myimage-20080710.raw</image>
    <byte_runs>
      <run fs_offset='31744' img_offset='64000' file_offset='0' len='2048'/>
      <run fs_offset='35840' img_offset='68096' file_offset='2048' len='14336'/>
    </byte_runs>
    <md5>3b4af47f8b542fb5d1bdaeec34563d89</md5>
    <sha1>b614d783fee50f053bd99da89401b89b013487a4</sha1>
  </original>
  <residual>
    <byte_runs>
      <run fs_offset='32256' img_offset='64512' file_offset='512' len='512'/>
      <run fs_offset='36352' img_offset='68608' file_offset='2560' len='512'/>
    </byte_runs>
  </residual>
</fileobject>
```

Fig. 9. How the XML from Figure 1 might be rewritten to indicate that only the second sector of each byte run is still resident on the disk.

We have developed a program that automatically builds a map of a final disk drive image using multiple previous images as forensic guides. The program's sole input is multiple disk images recorded from a single device at different times. The program, called `igroundtruth.py`, is approximately 200 lines of well-commented Python code (not including the `fiwalk` module). It follows a relatively straightforward algorithm:

1) The final disk image D_N is read and a single XML document is generated using the `fiwalk` module's `fileobjects_using_dom` function. All of the runs for all of the allocated files are stored in a *runs database*, a simple data structure which maintains a geometric map of the allocated sectors on the disk.
2) Each previous disk image $D_i = D_0 \ldots D_{N-1}$ is read, starting with the earliest. For each allocated file D_iF_j, the runs in the allocated file are compared with the runs in the runs database. If there is no intersection, the corresponding bytes for the runs are extracted from both D_i and D_N. If the extracted bytes from both disk images match, the residual data from file D_iF_j is still present in D_N. In this case, the XML for D_iF is copied into the document XML for D_N and annotated with a `<reference>` tag noting that it is residual data from image D_i.
3) All of the disk images are scanned a second time, except this time all of the individual sectors for each file D_iF are compared between images D_i and D_N. This allows the detection of residual data for which part of the file has been overwritten but part of the file remains. In this case, the XML for file D_iF is rewritten so that the original `<byte_runs>` are stored in a new tag called `<original>` and a new set of `<byte_runs>` are inserted into the XML indicating which sectors have been preserved.

An example of the resulting XML is shown in Figure 9.

B. Image Redaction Tool

In many cases it is desirable to explicitly remove information from a disk image before that image is handed over to another party. For example, forensic challenges sponsored by the Honeynet Project and DFRWS have not distributed disk images based on Microsoft Windows systems because of copyright

Redaction Conditions:

`FILENAME` *a file name*	A file with the given name
`FILEPAT` *a file pattern*	A file with the given pattern (e.g. *.txt)
`DIRNAME` *a directory*	All files in the given directory
`MD5` *an md5*	Any file with the given md5
`SHA1` *a sha1*	Any file with the given sha1
`FILE CONTAINS` *a string*	any file that contains *a string*
`SECTOR CONTAINS` *a string*	any sector that contains *a string*

Redaction Actions:

`FILL 0x44`	overwrite by filling with character 0x44 (Any hex code may be specified)
`ENCRYPT`	encrypts the data with the key specified in the configuration file.
`FUZZ`	Randomize the instructions of an executable, but leave the strings untouched.

Examples:

`FILEPAT *.DLL FUZZ`
Overwrite every byte of every DLL file with '`DDD`'... This is useful for forensic challenges and research projects that wish to distribute disk images of computers running Microsoft Windows. Fuzzing the DLLs breaks them so that they cannot execute, making it easier for those distributing the images to claim that such distribution is "fair use" under US copyright law.

`FILE CONTAINS steve@lawfirm.com ENCRYPT`
Encrypt every file that contains the string `steve@lawfirm.com`. This is useful for encrypting all files that may contain attorney-client privileged information.

`SECTOR CONTAINS frank@company.com FILL 0x44`
Fills every sector that contains `frank@company.com` with '`DDD`' This is useful for redacting all sectors that may contain residual information from attorney-client email.

Fig. 10. The redaction language used by `iredact`

concerns—this, despite the fact that Windows makes up the bulk of forensic practice. Likewise the movement of disk images in legal discovery is often problematic because the images may contain information that is attorney/client privileged.

We have developed `iredact`, a Python program that can redact information from a disk image based on a set of rules specified in a configuration file. The configuration file allows the user to specify specific conditions for redaction and specific redaction actions. The program then applies each rule to each file in the disk image and redacts accordingly. The current version of `iredact` supports seven redaction rules and three actions, both shown in Figure 10.

Despite being quite small (< 250 lines), `iredact` features an object-oriented plug-in architecture that allows the easy creation of new kinds of redaction rules and new kinds of methods. Plug-in systems like this are easy to create in Python.

C. USB Transfer Kiosk

Finally, we have used the `fiwalk` tool to create a stand-alone "transfer station" that allows users to copy data off removable devices such as USB memory sticks and external hard drives without actually mounting the device—allowing data to be cleanly transferred off devices that might be infected with viruses that propagate using `autorun.inf` and `desktop.ini` files. The USB Transfer Station does this by using `fiwalk` to obtain a list of all the files on the removable media, showing this list to the user, and then allowing the user to chose which files he or she wishes transferred. Each selected file is then copied sector-by-sector from its location on the removalbe storage, virus scanned, and finally copied to the user's network share.

The USB transfer station is a standard PC with an Intel processor running Ubuntu 8.10 Linux. A program called "watcher" scans the workstation for the insertion of new mass-storage devices by polling the `/dev/`

filesystem for new entries beginning `/dev/sd`. When a new entry is found, the watcher makes sure that the first sector of the device can be read. (Inserting a SD Card reader without an SD card will cause the `/dev/sd[a-z]` device to be created but it will not be readable until an SD card is actually inserted).

Once readable media is detected, an application called `kiosk.py` starts up which runs `fiwalk` using the SAX callback method; the callback function populates a File List. Meanwhile the user is asked for their username and password; these credentials are verified against an LDAP server.

When the user's credentials are verified the File List is enabled and the files from the USB device are displayed in the user interface. The user selects which files that he or she wishes to transfer to the organization's computer system. The user then selects the destination: their network home directory (as read from the LDAP server) or a "drop box" located on the transfer station.

When the user clicks the "go" button, the Transfer Station verifies the users credentials through a query against the LDAP server. It then copies the raw data for each selected file from the USB device to a special "quarantine" area, where the file is scanned using the ClamAV open source malware scanner.

Files that pass the AV scan are copied into the users home directory using the `smbclient` program, part of the Open Source "Samba" package. The smbclient program allows files to be transferred to a Windows file server without actually mounting the file server, similar to the way that files can be copied to an FTP server using the `ftp` command. In this way the Linux system never has unrestricted access to the users file server.

Alternatively, files can be copied into a "drop box," which is implemented as a webserver running on the Linux system. In this case the files are copied into a hidden directory and the user is sent a notification email with a URL that can be used to access the files. Files put on the drop box are automatically deleted after 72 hours.

All transfers are logged with the original file name, file size, original file date, MD5 and SHA1 hash code, and the intended destination. A future version of the kiosk will automatically encrypt files found to be containing viruses with a public key; the matching private key will not be not resident on the transfer kiosk. Once encrypted, these files will be taken to a forensic lab for further analysis.

Because the transfer kiosk never mounts the USB device, it is not susceptible to malware that spreads through mechanisms that automatically run code (e.g. Windows `autorun.inf` and `desktop.ini`, or that might exploit buffer overflows in rendering routines (e.g. in a JPEG rendering system that might automatically render icons on an inserted disk image.)

V. Conclusions

By writing a program that outputs XML associated with file system metadata, and by creating a python module that can ingest these XML files and present a simple object model to the user, we have made it relatively easy to write programs that can automatically process disk images and external storage devices.

Our experience presented here shows that it is relatively easy to take these building blocks and create tools that apply forensic capabilities to new domains. We believe that there are many applications that could benefit from the ability the ingest disk images in a forensically sound manner. Today the developers of these applications are put off by the difficulty in learning forensic concepts and working with the specialized APIs and data structures of SleuthKit or the obscure programming language of EnScript. We believe that the approach presented in this article is a good starting point to overcoming these barriers.

A. Availability

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

B. Future Work

It is our intention to integrate `fiwalk` with the SleuthKit Carrier's SleuthKit over the following months. We also hope to modify SleuthKit's `fls` program so that it can output compatible XML, to modify fiwalk so that it can also output in the SleuthKit legacy "mactime" format, and to modify modify SleuthKit's `mactime` program to read XML data to make timelines.

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

C. Acknowledgments

We wish to thank Jessy Cowan-Sharp, George Dinolt and Beth Rosenberg for their support of this project. This work was funded in part by National Institute of Standards and Technology the Naval Postgraduate School's Research Initiation Program. Thanks also to the anonymous reviewers, whose comments helped us to improve the quality of this paper.

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

REFERENCES

[1] Access Data. Forensic toolkit—overview, 2005. http://www.accessdata.com/Product04_Overview.htm?ProductNum=04.

[2] Brian Carrier. The Sleuth Kit & Autopsy: Forensics tools for Linux and other Unixes, 2005. http://www.sleuthkit.org/. [Online; accessed 06 March 2009].

[3] Brian Carrier, Eoghan Casey, and Wietse Venema. Dfrws 2006 forensics challenge overview, 2006. http://www.dfrws.org/2006/challenge/.

[4] Brian Carrier, Eoghan Casey, and Wietse Venema. Dfrws 2007 forensics challenge overview, 2007. http://www.dfrws.org/2007/challenge/.

[5] M.I. Cohen. Pyflag: An advanced network forensic framework. In *Proceedings of the 2008 Digital Forensics Research Workshop*. DFRWS, August 2008. http://www.pyflag.net. [Online; accessed 06 March 2009].

[6] Ian F. Darwin. Libmagic, August 2008. ftp://ftp.astron.com/pub/file/.

[7] Simson Garfinkel and David Malan. One big file is not enough: A critical evaluation of the dominant free-space sanitization technique. In *The 6th Workshop on Privacy Enhancing Technologies*, June 28 – 30 2006.

[8] Guidance Software, Inc. EnCase Forensic, 2007. http://www.guidancesoftware.com/products/ef_index.asp.

[9] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (dom) level 3 core specification, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

[10] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to privacy in the forensic analysis of database systems. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 91–102. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-686-8.