2012

Acquiring OS X File Handles through Forensic Memory Analysis

Andrew F. Hay
Center for Cyberspace Research, Air Force Institute of Technology

Gilbert L. Peterson
Air Force Institute of Technology

Recommended Citation

Hay, A., & Peterson, G. L. (2012). Acquiring OS X File Handles through Forensic Memory Analysis. UNESCO Systematic Approaches to Digital Forensics, 2012, 1–8.

Acquiring OS X File Handles Through Forensic Memory Analysis

Andrew F. Hay
ahay@ieee.org

Gilbert L. Peterson
gilbert.peterson@afit.edu

Center for Cyberspace Research
Air Force Institute of Technology
Wright-Patterson AFB, Ohio

Memory analysis has become a critical capability in digital forensics because it provides insight into system state that cannot be fully represented through traditional media analysis. The volafox open source project has begun the work of structured memory analysis for OS X with support for a limited set of kernel structures. This paper addresses one memory analysis deficiency on OS X with the introduction of a new volafox module for parsing file handles associated with running processes. The developed module outputs information comparable to the UNIX `lsof` (list open files) command, which is used to validate the results.

Keywords: forensics; memory analysis; Apple OS X; volafox; file handle

1. Introduction

This paper describes implementation of a new forensic capability for parsing open file information from OS X memory captures. When open files are mapped to a process, the forensic examiner learns which resources the process is accessing on disk. This listing is useful in determining what information may have been the target for exfilitration or modification on a compromised system. File handles may also help identify a suspicious process when unexpected file access or modifications are observed. Carvey further describes how a list of open files can compliment disk analysis to identify files of interest during an investigation[1]. Because open files can help characterize process activity and highlight misuse of a computer, it is highly desirable to recover this information from memory.

To support the extraction of file handles from raw memory two research objectives are defined. These include the design recovery of kernel data structures responsible for handling open files, and development of a flexible process for programmatically handling structures defined for different kernel architectures and operating system (OS) versions. This necessitates extensible software design resilient to changes in future versions of OS X.

[1] Carvey, Harlan. 2009. *Windows forensic analysis DVD toolkit* (2nd ed.). Burlington: Syngress.

Project volafox offers an open source memory analysis solution for OS X and FreeBSD written in Python[2]. Revision 52 of the source code has support for a limited set of kernel structures to parse hardware information, system build number, process listing, socket connections, loaded kernel extensions, and the syscall table. The project is extended by this research with the design recovery, structure template process, and new volafox module for parsing file handles presented in Section 3.

To validate the handles module, the UNIX command line tool `lsof` (list open files) is used to baseline the state of open files for comparison. Testing was accomplished in a controlled environment using four virtual installations of OS X across two OS versions (10.6 Snow Leopard, 10.7 Lion) and two kernel architectures (i386, x86_64). Output from the `lsof` command is compared with the handles module using an automated script that classifies differences according to a taxonomy in order to filter the results for further analysis.

2. Background

The memory forensics process consists of two parts. First, a copy of the target's memory called an *image* is written to external media. This requires a toolkit consisting of software to perform the capture and a USB device or network connection to preserve the image. Second, the image is analyzed on a forensic workstation using tools to extract human-interpretable information.

2.1. Mac Memory Acquisition

Two imaging methods for OS X are considered in this research. First, the memory backup file saved by the host system of an OS X virtual machine (VM) during suspension can be copied to image the guest memory[3]. However, due to tight hardware-software integration on the Mac it would be rare to encounter such an installation in the field, thereby limiting its usefulness to the forensic investigator.

---

[2] Lee, Kyeongsik. 2012. "volafox: Memory analyzer for Mac OS X & BSD." Accessed 24 May. https://code.google.com/p/volafox/

[3] Ligh, Michael, Steven Adair, Blake Hartstein, and Matthew Richard. 2011. *Malware analyst's cookbook and DVD: Tools and techniques for fighting malicious code*. Indianapolis: Wiley.

Suiche[4] demonstrates a second method using emulation of `/dev/mem` to dump RAM after retrieving critical symbols needed to build a kernel memory manager. This capability is available as an OS X kernel extension with the Mac Memory Reader tool[5]. There are several disadvantages to this form of acquisition. First, loading the kernel module needed to browse full memory address space requires administrator privileges. Second, output from such a tool could be corrupted by the presence of memory forensic countermeasures[6] or advent of a rootkit explicitly designed to subvert collection. "Fortunately, unless the subversion mechanism is very deeply embedded in the OS, a substantial amount of overhead may be incurred to prevent acquisition, potentially revealing the presence of a malicious agent"[7]. Finally, because software must be executed on the target to perform the capture, its use alters system state. Despite the disadvantages, availability of this robust acquisition capability for the Mac encourages additional research and emphasis on analytic capabilities for the platform.

2.2. Project Volafox

This file handle parsing research adds a module to the existing open source volafox project[8]. Figure1 shows a summary of the source files from the volafox package relevant to OS X memory analysis with public classes in bold. Connections represent file dependencies, which are labeled with the public function names. The new open files module (`lsof.py`) is shown but not discussed until Section 3.

The project `main()` and volafox class found in `volafox.py` are responsible for marshaling the remaining files and classes to perform memory analysis. This source file interfaces with the new file handles module (`lsof.py`), and a number of other files indirectly related to its functionality.

---

[4] Suiche, M. 2010. Mac OS X physical memory analysis. *Black Hat DC.* http://www.blackhat.com/presentations/bh-ds-10/Suiche_Matthieu/Blackhat-DC-2010-Advanced-Max-OS-X-Physical-Memory-Analysis-wp.pdf

[5] Architecture Technology Corporation. 2011. "Mac Memory Reader." Accessed 24 May. http:// cybermarshal.com/index.php/cyber-marshal-utilities/mac-memory-reader

[6] Haruyama, Takahiro, and Hiroshi Suziki. 2012. "One-byte Modifications for Breaking Memory Forensic Analysis." Slides presented at Black Hat Europe, Amsterdam, Netherlands, March 14-16. https://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_ Forensic-Slides.pdf

[7] Case, Andrew, Andrew Cristina, Lodovico Marziale, Golden Richard, Vassil Roussev. 2008. "FACE: Automated digital evidence discovery and correlation." *Digital Investigation* 5(Supplement): S65-S75.

[8] See 2

The `x86.py` and `ia32_pml4.py` files house the address space agnostic classes `IA32PagedMemoryPae` and `IA32PML4MemoryPae` respectively. They are responsible for performing virtual to physical address translations that can subsequently be converted to file offsets by `FileAddressSpace`. All requests for reading raw memory are passed through one of these two objects. PML4 is a reference to the 4th level page map used by the Intel IA-32e paging scheme[9], meaning `ia32_pml4.py` is responsible for handling 64-bit images where `x86.py` is used for 32-bit.

Kernel architecture is determined using the `imageInfo` class, which also returns the OS build version. This number is needed to select the correct overlay file containing the symbol list for a particular version of OS X. All symbols are read from files in the `overlays` directory using the Python `pickle` library for object serialization. New overlays can be generated from the kernel executable (`mach_kernel`) using the `overlay_generator.py` utility.

Revision 52 of volafox, the version extended for this research, does not natively support the Mac Memory Reader (MMR) output format. Leat[10] and ATC developer Hajime Inoue contributed experimental support for MMR which is operational in revisions 23-38 on the project website. The feature was later removed with the introduction of 64-bit addressing support due to compatibility problems. A stand-alone `flatten.py` utility authored by Inoue is still available to convert MMR files to a linear format, but only works for 32-bit kernel installations. This utility was employed to analyze the real-world memory captures discussed in Section 4.3.

2.3. Structured Memory Analysis

Most references to memory analysis on OS X discuss context-free techniques such as string searches, manual hex examination, and file carving[11]. In order to perform meaningful analysis of a memory image, an understanding of the composition and location of key kernel

---

[9] Intel Corporation. 2012. "Intel 64 and IA-32 Architectures Software Developer's Manual." Accessed 31 March. http://download.intel.com/products/processor/manual/325462.pdf

[10] Leat, Chris. 2011. "Forensic analysis of Mac OS X memory." Master's thesis, University of Westminster.\

[11] Makinen, Juho. 2008. "Automated OS X Macintosh password retrieval via FireWire." Last modified February 29. http://www.juhonkoti.net/2008/02/29/ automated-os-x-macintosh-password-retrieval-via-firewire
Malard, Arnaud. 2011. "H@CKMacOSX: Tips and tricks for Mac OS X hack." Last modified August 3. http://sud0man.blogspot.com/2011/08/hackmacosx-tips-and-tricks-for-mac-os-x.html

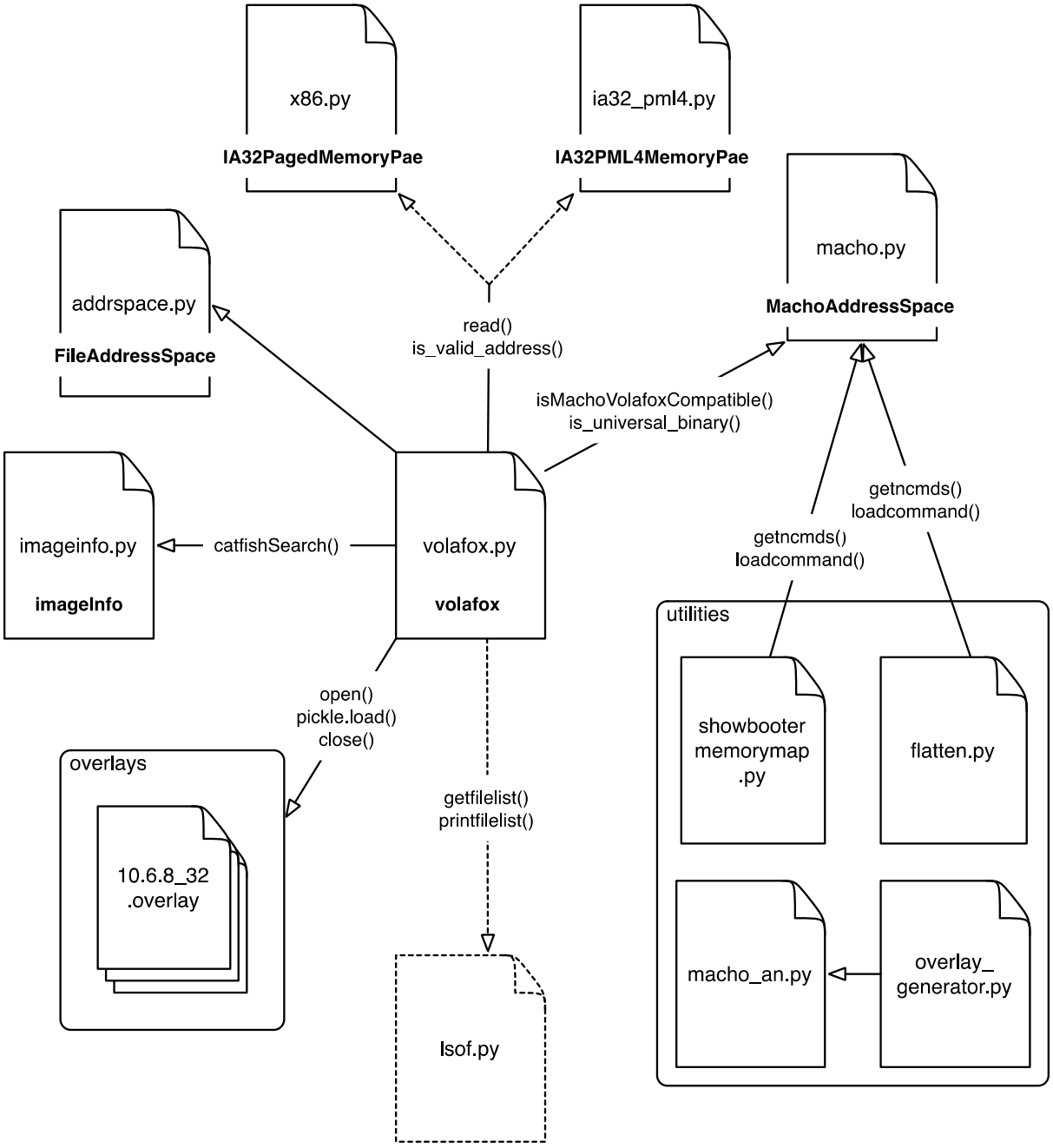

Figure 1. volafox release 52 package diagram.

structures is required. Because Darwin (Apple's UNIX core for OS X) is open source, the composition of kernel structures can be determined from the header files they are defined in.

Locating the structures in memory requires a mapping of identifiers and offsets, or a kernel symbol table. Suiche notes "[s]ymbols are a key element of volatile memory forensics

without them an advanced analysis is impossible"[12]. The KPCR structure can be used in Windows to get the symbols directly from memory[13], however in OS X the equivalent "kernel sections are destroyed as soon as the kernel (`mach_kernel`) is loaded"[14]Volatility solves this problem using a database of overlay files containing the requisite symbol tables for select Linux distributions and kernel versions[15], similar functionality is provided for volafox by Leat[16]. In both Linux and OS X therefore, the "easiest way to retrieve kernel symbols is to extract them from the kernel executable of the hard-drive"[17].

Figure 2 shows key features of the `mach_kernel` executable, located at the root directory of the OS X file system. Suiche[18] describes how knowledge of the file's structure can be used to build a symbol table for a particular build of OS X. Figure 3 demonstrates how the symbol table derived from the `mach_kernel` executable is used to parse a list of running processes. Symbol `_kernproc` provides a static address for the head of the process list, kernel_task (PID 0), which is unique in its use of static data structures (Singh 2006, 293). The process ID, parent's PID, command name and pointer to `struct pgrp` are members of `struct proc`. Associated username information is located in `struct session`, which is referenced by `struct pgrp`. The substructure `p_list` provides a linked-list that can be walked to parse the entire running process list.

3. Methodology

[12] ibid 4

[13] Dolan-Gavitt, Brendan. 2008. "Finding kernel global variables in Windows." Last modified April 16. http://moyix.blogspot.com/2008/04/finding-kernel-global-variables-in.html

[14] ibid 4

[15] Case, Andrew. 2011. "Bringing Linux support to Volatility." Last modified March 1. http://dfsforensics.blogspot.com/2011/03/ bringing-linux-support-to-volatility.html

[16] ibid 10

[17] ibid 4

[18] ibid 4

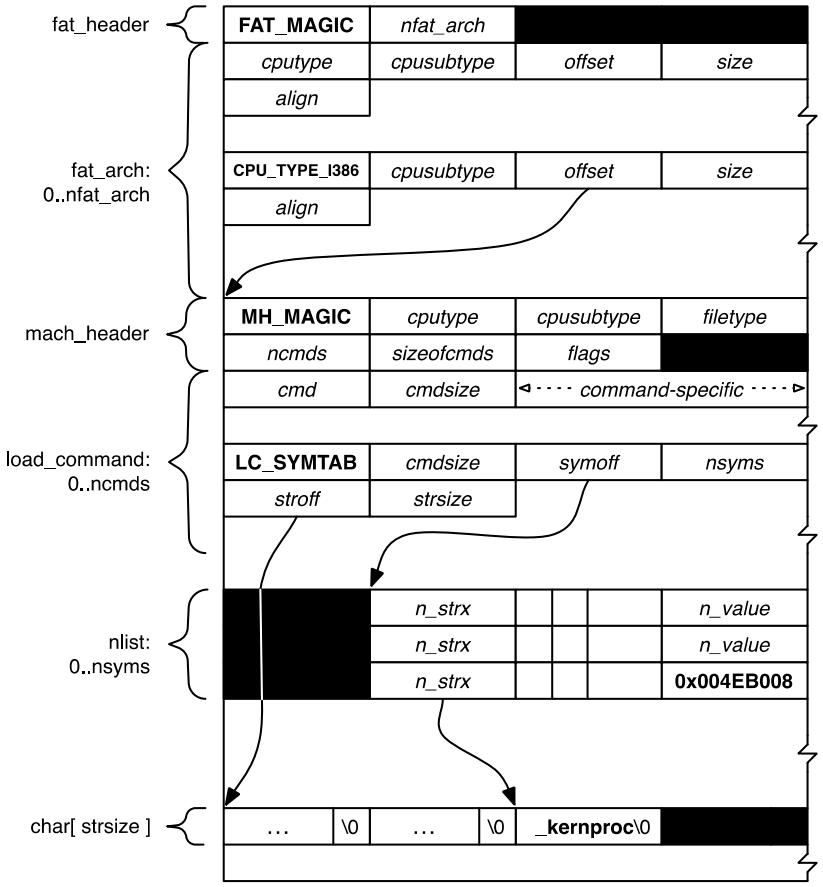

Figure 2. `mach_kernel` executable.

The desired process-to-file handle information is an approximation of output from the UNIX `lsof` command for OS X[19] (Apple Inc. 2011). Because `lsof` is included with operating system, it offers a reliable source of information for comparison. Emulating the output of this tool provides validation and offers the examiner a familiar interface to interact with. Figure 4 shows sample `lsof` output and Table 1 describes the information in each column. The new volafox module for listing file handles includes functionality for parsing the nine default `lsof` fields and the mode identifier integrated with the FD column.

While an ideal implementation would fully duplicate functionality of the `lsof` command, due to the diversity of data structures involved the problem must be scoped for this research. Two design decisions bounding the implementation are the subset of handle types supported, and the subset of filesystems supported.

[19] Apple Inc. 2011. "lsof(8) Mac OS X manual page." Last modified March 21. http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man8/lsof.8.html

The volafox open files module supports handle types that subscribe to the virtual node (vnode) interface. Excluded types include POSIX semaphores and shared memory files, kernel event queue files, pipes, and sockets. These types are reported as part of the file descriptor table, but with DEVICE, SIZE/OFF, NODE and NAME fields unsupported. Additionally, the UNIX `lsof` command classifies sockets by a variety of subtypes, which the volafox open files module groups together using the generic type description '`SOCKET`'.

Filesystem support includes HFS+ and DEVFS. HFS+ is the default format for the OS X boot volume and DEVFS is used to abstract certain devices such as special character files. Among other uses, special character files describe `ttys` devices controlling the print streams `stdin`, `stdout`, and `stderr` for terminal programs. HFS+ and DEVFS account for the filesystems most commonly encountered during development and testing, but the vnode interface makes reference to at least 20 other types. One impact of this constraint is that files stored on network filesystems, FAT32, NTFS and others, do not have volafox support for `lsof` fields outside the vnode interface.

3.2. Kernel Design Recovery

Developing the new volafox module for listing open files requires an understanding of 32 unique C data structures from the OS X source code, four of which are described by Suiche[20] to list running processes. These include 26 structure (struct), three enumeration (enum), and three union definitions. Identifying the data structures containing critical information and the relationships between them is one of the primary contributions of this research because "the

```
$ lsof –p 109
COMMAND  PID  USER  FD  TYPE  DEVICE  SIZE/OFF   NODE NAME
bash     109  6ad  cwd   DIR   14,2        578 202041 /Users/6ad
bash     109  6ad  txt   REG   14,2    1346544 262558 /bin/bash
bash     109  6ad  txt   REG   14,2    1054960 264388 /usr/lib/dyld
bash     109  6ad  txt   REG   14,2  213385216 466405 /private/var/db/
                                                      dyld/dyld_shared
                                                      _cache_x86_64
bash     109  6ad    0u  CHR   16,0      0t369    611 /dev/ttys000
bash     109  6ad    1u  CHR   16,0      0t369    611 /dev/ttys000
bash     109  6ad    2u  CHR   16,0      0t369    611 /dev/ttys000
bash     109  6ad  255u  CHR   16,0      0t369    611 /dev/ttys000
```

Figure 4. UNIX list open files (`lsof`) command.

[20] ibid 4

Table 1. UNIX `lsof` output fields.

COMMAND	First nine characters of the UNIX command associated with the process.
PID	Process identification number.
USER	Login name of the user to whom the process belongs.
FD	File descriptor is a numeral index into the process open handle array optionally followed by a mode identifier: `r` (read access), `w` (write access), or `u` (both). Two other descriptors commonly seen are `cwd` representing the current working directory for the process and `txt` used for program text (code and data). These files are of high forensic value because they include the executable from which the command was launched, linked libraries, and other memory-mapped files. See the output section of the `lsof` manpage for a full list of descriptors used (Apple Inc., 2011).
TYPE	Node type associated with the handle. See the output section of the `lsof` manpage a partial type listing (Apple Inc., 2011). Note that numerous undocumented types were encountered in testing such as FSEVENT.
DEVICE	Major and minor device numbers separated by a comma. The first number describes a class of hard/software device and the second is a unique identifier for a particular instance of that class.
SIZE/OFF	Size or offset of a file reported in bytes. Offsets are preceded by a leading `0t` to distinguish when the column is mixed.
NODE	The node number for a local file. This unique identifier is filesystem dependent. For example, files stored on HFS+ report the catalog node identifier (CNID) for this field, whereas DEVFS files use a UNIX inode number instead.
NAME	Mount point and file system on which a file resides, or name of character special device.

member associated with each `lsof` field is shown in Table 2, however a few additional details are needed to understand the linked data structures and data decoding.

Structure `task`, as pointed to by `proc` in Figure 6, provides a link to program text files (FD `txt`). Note that each memory object may reference a `struct vm_object` or recursively refer to another entry. Memory mapped files are backed by a vnode pager, but the pager may be located in the shadow object for external memory managers (Singh 2006, 571).

The file descriptor table and current working directory are referenced from `struct filedesc` as shown in Figure 7. Member `filedesc.fd_ofiles` is a pointer to the start of a `fileproc` array. Elements of the array that contain a valid `fileglob` pointer reference a handle, those that do not are available to hold one. The array index represents the numerical file identifier used by the FD field of the `lsof` output. The integer `filedesc.fd_lastfile` indexes the last file in the array and provides an iteration bound. The array itself makes up the file descriptor table, used by a process to reference all open files (ASCII, word processing, logs, temp, etc.). The file mode, also known as read/write access, is determined from the value of `fileglob.fg_flag` using the bitmap definitions in `bsd/sys/fcntl.h`.

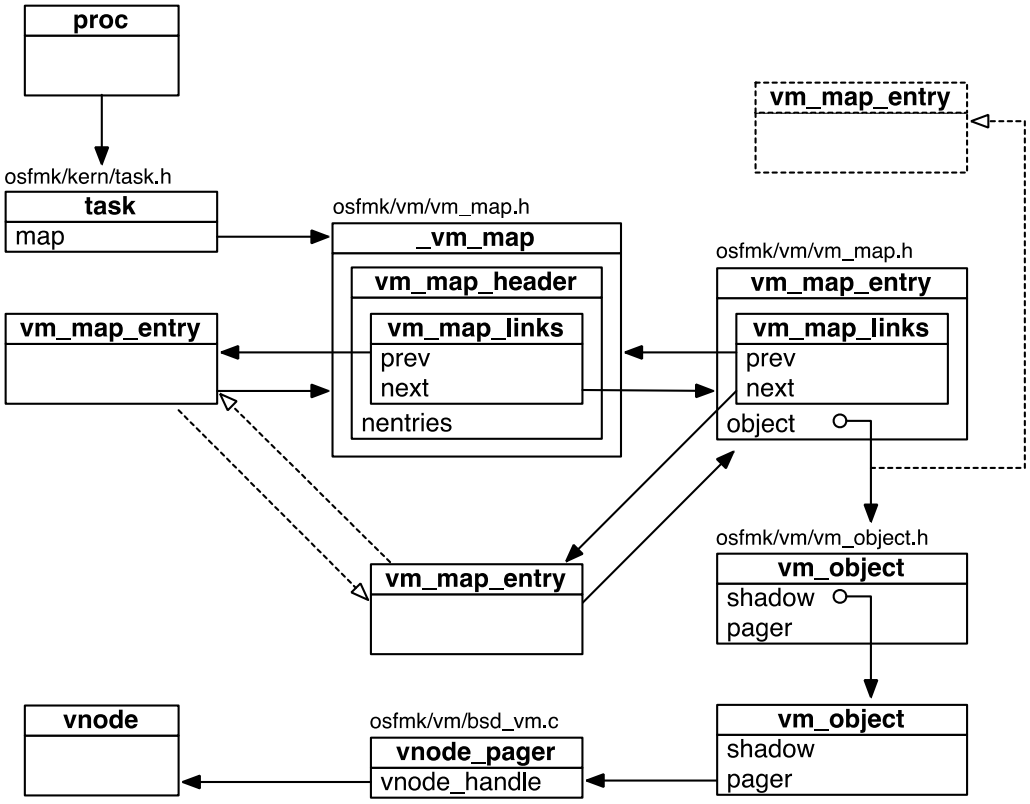

Figure 6. Memory-mapped files (`txt`).

Determining the handle device involves additional logic because the values are encoded. The device identifier parsed from either `struct mount` or `struct specinfo` (depending on the handle filesystem) is decoded using macros in `bsd/sys/types.h` to return major and minor device number. Similarly, returning the correct SIZE/OFF value for directories requires calculation using a count value found in a substructure of `struct cnode`, the equation:

$$(entries + 2) * \text{AVERAGE\_HFSDIRENTRY\_SIZE}$$

found in `bsd/hfs/hfs_vnops.c`, and the macro definition from `bsd/hfs/hfs.h`.

3.3. Structure Templates

Existing volafox modules are not readily extensible and require additional logic branching for each variant in size or composition of the underlying kernel data structures. This section documents a solution developed for flexible analysis of multiple kernel architecture and OS versions. The solution consists of two parts. First, an interface is specified to describe data structures called templates. Next, a dynamic mechanism using software classes is described for

Table 3. Template interface fields.

Variable	Python Type	Description
`template`	dict	template implementing the C stuct interface
`MBR_NAME`	str	dictionary key, variable name for a struct member
`template[MBR_NAME]`	tuple	dictionary value, a struct member description
`MBR_TYPE`	str	C type of the named member
`OFFSET`	int	offset in bytes for the member
`SIZE`	int	size in bytes for the member type
`FIELD`	str	lsof field represented by member
`SUB_STRUCT`	dict	recursively defined substructure (*optional*)

selecting the correct template for a particular memory image based kernel architecture and OS version at runtime. A method for automated generation of the templates is also presented.

3.3.1 Template Interface.

The following interface is defined for required members of each structure:

```
template = { MBR_NAME : ( MBR_TYPE, OFFSET, SIZE, FIELD,
                          SUB_STRUCT ), … }
```

Table 3 lists Python types from the kernel structure template interface, which itself is implemented as a dictionary. Substructures are defined as those contained within the memory allocated for a super structure. They share the same dictionary format as regular structures and their values are referenced recursively. Figure 8 shows the 32-bit Snow Leopard variant of the `struct proc` template as an example. To support the test cases described in Section 4.2, three additional templates for the process structure are defined, one for each combination of OS version and architecture.

```
template = {
    'p_list' : ( 'LIST_ENTRY(proc)', 0, 8, '' , {
        'le_next' : ( 'struct proc *', 0, 4, '' ),
        'le_prev' : ( 'struct proc **', 4, 4, '' )
       }
    ),
    'p_pid' : ( 'pid_t', 8, 4, 'PID' ),
    'task' : ( 'void *', 12, 4, '' ),
    'p_fd' : ( 'struct filedesc *', 104, 4, '' ),
    'p_textvp' : ( 'struct vnode *', 388, 4, '' ),
    'p_comm' : ( 'char[]', 420, 17, 'COMMAND' ),
    'p_pgrp' : ( 'struct pgrp *', 472, 4, '' )
}
```

Figure 8. `struct proc` template, for Mac OS X 10.6 on x86 hardware.

3.3.2 Template Selection.

The second component in the template solution is a Python class initializer that dynamically selects the correct template for a given subclass based on the OS version and architecture of the memory image under analysis. Because classes in the open files module manage fields and methods associated with a particular kernel structure, all inherit from the abstract superclass in Figure 9.

The first four static variables belong to the abstract class and are shared by all `Struct` subclasses. The `mem` variable is a reference to one of the PAE objects responsible for virtual-to-physical address translation. Verbose flag `ver` indicates if *all* file descriptors should be printed, including those for types not fully supported by the open files module. The `arch` and `kvers` variables report the kernel architecture and version respectively. The final three fields are virtual static variables because their assignment is deferred to the subclasses. The constant `TEMPLATES` is a nested dictionary from which the static `template` is assigned the first time the initializer runs based on value of `arch` and `kvers`. The static `ssize` is subsequently assigned based on the selected template and determines how many bytes the initializer reads from the address passed as an argument to provide coverage of all members specified in the structure template.

```
class Struct(object):

  mem       = None
  ver       = False
  arch      = -1
  kvers     = -1

  TEMPLATES = None
  template  = None
  ssize     = -1

  def __init__(self, addr):

    if self.__class__.template == None:

      self.__class__.template = self.__class__.TEMPLATES[Struct.arch] \
                                                         [Struct.kvers]

      for item in self.__class__.template.values():
        if ( item[1] + item[2] ) > self.__class__.ssize:
          self.__class__.ssize = item[1] + item[2]

      self.smem = Struct.mem.read(addr, self.__class__.ssize);
```

Figure 9. Simplified abstract `class Struct` (no error handling).

```
class Devnode(Struct):

  TEMPLATES = {
    32:{
      10:{'dn_ino':('ino_t',112,4,'NODE')}
      , 11:{'dn_ino':('ino_t',112,4,'NODE')}
    },
    64:{
      10:{'dn_ino':('ino_t',192,8,'NODE')}
      , 11:{'dn_ino':('ino_t',192,8,'NODE')}
    }
  }

  def __init__(self, addr):
    super(Devnode, self).__init__(addr)

  def getnode(self):
    return unpacktype(self.smem, self.template['dn_ino'], INT)
```

Figure 10. Concrete class Devnode.

Combining the structure template interface with an abstract initializer offers a solution that greatly simplifies program logic needed to support a selection of architectures and OS versions. The result is also highly extensible because new templates can be added without any code refactoring as long as the member names remain consistent across versions. Figure 10 shows the concrete subclass corresponding to `struct devnode` and demonstrates use of the structure template solution.

3.3.3 Member Offsets and Type Sizing.

While the dictionary constants used to implement structure templates are easy to work with programmatically, generating their syntax is labor intensive. The new open files module uses *(18 classes * 2 versions * 2 architectures) = 72* structure templates, requiring a great deal of error-prone coding and debugging if generated by hand. Determining *size* and *offset* values for each member in the template is also very difficult to accomplish manually due to the complexity of defined types included in the kernel structures. The solution to both of these challenges is an external C program that dissects kernel structures and automates the generation of the Python dictionary syntax needed for each template.

The `offsets.c` program was developed to find the size and offset of each required structure member and print the results as a structure template for use in `lsof.py`. Figure 11 shows a function from the program that prints a template for `struct _vm_map`. The variable `member` is a C structure defined in the program to hold the fields described in Table 3 and

```
int vm_map() {

	member m;
	int (*mh)(unsigned long int offset) = &vm_map_header;

	printf("struct_vmmap = {");

	m.var_name = "hdr";
	m.var_type = "struct vm_map_header";
	m.offset = offsetof(struct _vm_map, hdr);
	m.size = sizeof(struct vm_map_header);
	m.field = "";
	printmember(m, mh);

	printf("}\n");
	return 0;
}
```

Figure 11. Template generation function.

`printmember()` formats each as a key/value pair for the enclosing Python dictionary. The argument `mh` is a function pointer to a substructure that is printed recursively.

The C language `sizeof` operator is used to find the size of any type, and the preprocessing macro `offsetof` defined in `stddef.h` can return the offset of any member for a given structure. However, most of the header files defining kernel structures are not available in the include path for OS X. Sesek (2012) explains the problem and suggests a workaround in a blog post about kernel debugging:

> Structs […] are merely human-friendly offsets into a region of memory. Their definition and layout can be shamelessly copied from the XNU open source headers into your kext's project so that you can access fields in kernel private structures. As it turns out, virtually ever structure within the kernel is designed to be opaque to a kext. Apple decided to do this so that they can freely change the kernel structures, but it also makes writing a debugging tool like this a little harder. To do so you need to edit the headers so they compile in your project through a process I call "munging."

Sesek's method was modified to access the kernel definitions needed for `offsets.c` using local headers.

Three out of 18 template functions written for `offsets.c` are known to produce incorrect member offsets for 64-bit kernel architecture. The problem is believed to be a complex definition conflict for some low-level types. Several C types are defined for userspace with standard libraries such as `stdio.h`. However, the kernel sometime uses different sizes for these same types and forced redefinition yields a compilation error. When the `offsetof` macro

measures a userspace definition the result is an error for some architectures. Manual offset calculation and hex analysis are used to resolve the problem for affected templates, resulting in adjustment made to the `TEMPLATES` constant of the equivalent structure class in `lsof.py`.

A wrapper for `offsets.c` called `printstructs.py` is written to verify the output dictionary as executable Python code, print the structure members in a human-readable format for debugging, and handle compilation flags related to architecture. Dictionary output from `printstructs.py` was then pasted into the `TEMPLATES` constant of each class in `lsof.py` to complete the definition.

3.4. Open Issues

There are two outstanding problems with the research module developed: reporting the correct user associated with a process, and determining size of the `/dev` directory.

The manpage for the UNIX `lsof` command describes output of the USER field as "the user ID number or login name of the user to whom the process belongs, usually the same as reported by `ps(1)`". However, output from the volafox open files module is known to incorrectly report the process login name as shown in Figures 12-13.

The difference shown is not consistent across all processes of a full file listing. In many cases the expected USER value is reflected in the output, but not always. This problem is not unique to the volafox open files implementation as it is also present in the volafox process listing module and the original work on which it was based (Suiche 2010). Kernel structure analysis of the source headers could not identify an issue with Suiche's methodology, but all tests indicate that `struct session` cannot consistently return the username for any of the user-related

```
$ ./volafox.py -i 10.6.8x86.vmem -o lsof -p 15
COMMAND   PID   USER   FD      TYPE […]
distnoted  15   root  cwd       DIR […]
distnoted  15   root  txt       REG […]
distnoted  15   root  txt       REG […]
distnoted  15   root  txt       REG […]
distnoted  15   root    0r      CHR […]
distnoted  15   root    1      PIPE […]
distnoted  15   root    2      PIPE […]
distnoted  15   root    3u   KQUEUE […]
distnoted  15   root    56u  SOCKET […]
```

Figure 12. volafox user output.

```
# lsof -p 15
COMMAND   PID     USER […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
distnoted  15   daemon […]
```

Figure 13. `lsof` user output.

keywords available for `ps`. There is also no known method to determine when the `session` structure returns the correct value.

A second problem identified during development is an inability to correctly report the SIZE/OFF field for certain directories. The `/dev` directory is typed `DTYPE_VNODE` in `fileglob.fg_type` and `VDIR` in `vnode.v_type`. However, it has a tag of `VT_DEVFS` from `vnode.v_tag` rather than the `VT_HFS` seen for most other directories. Figure 14 shows an example of `/dev` as reported by the UNIX `lsof` command.

```
# lsof +d /dev
COMMAND PID USER   FD   TYPE     DEVICE SIZE/OFF NODE NAME
launchd   1 root    8r    DIR 20,5853800     4495  305 /dev
```

Figure 14. `/dev` directory size.

Note that $4495 \bmod 34 \neq 0$, and therefore sizing by the entry count as described in Section 3.2 is not valid for this directory. **Error! Reference source not found.**2 includes three alternate locations for the size applicable to other file types, but none were found to be effective in this case. Fortunately, due to the unique combination of tag and type for `/dev`, the failure is possible to detect. Since the location of the size is unknown, the volafox open files module prints `-1` for the size of `/dev` to indicate the field is unsupported.

4. Testing

Testing effectiveness of the volafox module for listing OS X file handles involves comparing its output with that of `lsof`. A successful implementation of the system must accurately report all file handles, adjusted for stated constraints and known deficiencies. Testing is conducted on controlled test cases and on captures from real user's machines.

The complex nature of a modern operating system like OS X guarantees changes to the system state between the time when the `lsof` command is run and the memory dump occurs[21]. Some allowance is necessary to account for volatility of the handles list during this interval. A successful implementation therefore becomes one that can be validated against the UNIX `lsof` command, adjusted for stated constraints, known deficiencies, and accuracy of the validation method.

[21] Hay, Brian, Kara Nance, and Matt Bishop. 2009. "Live analysis: Process and challenges." *IEEE Security & Privacy* 7(2): 30-37.

Table 4. Field differences versus file type.

File Type	COMMAND	PID	USER	FD+ mode	TYPE	DEVICE	SIZE/ OFF	NODE	NAME
cwd	✓	✓	D1	✓	✓	✓	✓	✓	✓
txt	✓	✓	D1	✓	✓	✓	✓	✓	✓
REG	✓	✓	D1	✓	✓	✓	✓	✓	✓
DIR	✓	✓	D1	✓	✓	✓	D2	✓	✓
CHR	✓	✓	D1	✓	✓	✓	E7	✓	✓
LINK	✓	✓	D1	✓	E5	✓	✓	✓	✓
FIFO	✓	✓	D1	✓	✓	E6	✓	✓	✓
VNODE (other)	✓	✓	D1	✓	✓	C3	C3	C3	✓
PSXSHM	✓	✓	D1	✓	✓	C2	C2	C2	C2
PSXSEM	✓	✓	D1	✓	✓	C2	C2	C2	C2
KQUEUE	✓	✓	D1	✓	✓	C2	C2	C2	C2
PIPE	✓	✓	D1	✓	✓	C2	C2	C2	C2
FSEVENT	✓	✓	D1	✓	✓	C2	C2	C2	C2
SOCKET	✓	✓	D1	✓	C1	C2	C2	C2	C2

4.1. Comparison Taxonomy

A formal list of 21 differences across four categories is used to classify reasons output from `lsof` and the new volafox handles module may differ. Taxonomic categories consist of constraints, deficiencies, explained differences, and failures. Enumeration labels are employed by the script `validate.py` to describe how similar the volafox output is to its validation data.

4.1.1 Constraints.

Constraints are defined as differences in output that occur due to system design decisions. The volafox open files module has several limitations with regard to handle type and filesystem tag that are used to scope the research implementation.

C1. The `lsof` subtype for socket handles cannot be determined. A value of `DTYPE_SOCKET` for the member `filglob.fg_type` indicates a socket handle. The `lsof` command reports a number of subtypes for these handles including: `systm`, `unix`, `IPv4`, `IPv6`, `rte`, `key`, `ndrv`, and possibly others that were not observed in testing. Sockets are assigned the generic type `SOCKET` in the volafox open files output.

C2. Only handles subscribing to the virtual node (vnode) interface are fully supported. A value of `DTYPE_VNODE` for the member `fileglob.fg_type` indicates the vnode interface is in use for a particular handle. Full support indicates meaningful output is reported for all nine `lsof` command fields. Non-vnode handles show the value '`-1`' for DEVICE, SIZE/OFF, NODE, and NAME to indicate these fields are unsupported in the volafox open files output.

C3. Only vnodes tagged HFS+ or DEVFS are fully supported. A value of `VT_HFS` or `VT_DEVFS` for the member `vnode.v_tag` indicates a supported filesystem. The `lsof` command fields DEVICE, SIZE/OFF, and NODE are defined outside `struct vnode` and therefore unsupported for other filesystems. Unsupported fields are indicated in the volafox open files output with an appropriate value from `ECODE`, a global dictionary defined for `lsof.py`.

4.1.2 Deficiencies.

Deficiencies are defined as differences in output that occur due to known implementation problems. As described in Section 3.4, the volafox open files module has two open issues.

D1. The `lsof` USER field is not correctly reported for all processes in a full file listing. This problem is not consistent across all processes and the volafox open files module is not capable of detecting its occurrence.

D2. Size of the `/dev` directory cannot be determined. Handles with `vnode.v_type` of `VDIR` and `vnode.v_tag` of `VT_DEVFS` such as `/dev` show the value '`-1`' in the SIZE/OFF field.

4.1.3 Explanations.

Explained differences are those in output that occur due to reproducible idiosyncrasies of the tools used for capture or validation. They are distinct from failures because the explanations are not speculative, and the differences can be detected using automation. Explanations E4, E5, and E6 are believed to be bugs in the OS X version of the `lsof` program.

E1. The UNIX `lsof` command output always includes the `lsof` command and its associated handles, whereas a memory dump does not. For 10.7 only, the dependent process `sudo` is present in addition to `lsof` when executed with administrator privileges.

E2. Memory captured using the MMR tool includes handles associated with the process `MacMemoryReader` and its dependency `image`, whereas output from the `lsof` command does not.

E3. Data collected using `capture.py` (4.3) does not share the process `sh` because `MacMemoryReader` and `lsof` are executed in different subprocesses.

E4. OS X duplicates some handles in a full listing using `lsof`. Duplication occurs at least once per listing. Figure 15 demonstrates the problem.

```
$ sudo lsof
COMMAND PID   USER    FD    TYPE   DEVICE  SIZE/OFF   NODE NAME
...
mds      29   root   cwd     DIR     14,2      1088      2 /
mds      29   root   twd     DIR     14,2      1088      2 /
...
```

Figure 15. `lsof` handle duplication.

In all observed cases, the file descriptor '`twd`' (the per-thread working directory) identifies the duplicate, while all other fields remain the same.

E5. OS X reports the type of symbolic links as '`0012`' instead of '`LINK`' in the `lsof` TYPE field. The keyword '`LINK`' is specified in the manpage and therefore the volafox handles module reports symbolic links using that label. The bug has only been observed in the 10.7 version of OS X.

E6. OS X does not report the `lsof` DEVICE field for FIFO type files. The manpage does not discuss the omission and the volafox open files module can determine the major and minor device number for FIFO special files.

E7. Execution of the `lsof` command causes the offset of its terminal file (`ttys`) to grow. For cases where a `ttys` file is the same used by the `lsof` command, any offset difference is classified as E7 rather than F6.

4.1.4 Failures.

Failures are defined as differences in output not already accounted for by constraints, deficiencies, or explanations that occur due to asynchronous data collection or implementation artifact. It is important to note that the fault *causing* failure is undefined by default. Analysis in Section 4.2 indicates that in most cases failure is a consequence of validation accuracy rather than an error in the volafox open files module implementation.

F1. Command name mismatch (field: COMMAND). Adjusted for F2.

F2. Missing/extra process (field: PID). Adjusted for E1, E2, and E3.

F3. Missing/extra file descriptor (field: FD). Adjusted for F2 and E4.

F4. File mode mismatch (field: FD). Adjusted for F3.

F5. File type mismatch (field: TYPE). Adjusted for F3, C1 and E5.

F6. Device mismatch (field: DEVICE). Adjusted for F3, C2, C3, and E6.

F7. Size/offset mismatch (field: SIZE/OFF). Adjusted for F3, C2, C3, D2, and E7.

F8. Node identifier mismatch (field: NODE). Adjusted for F3, C2 and C3.

F9. Pathname mismatch (field: NAME). Adjusted for F3, C2.

Username mismatch is classified as D1 and therefore not listed as a failure. It is reported in the results after adjustment for F2. Reporting failures F2 and F3 also aligns the process and handle lists of each file respectively for the remaining failure tests. This means, for example, that F1 does not report command name mismatches that occur due to a missing process because F2 already accounts for it.

4.2. Controlled Test Cases

Controlled test case results are examined with the goal of identifying previously unidentified implementation problems. The majority of constraints, deficiencies, and explained differences are not considered in this analysis as the failures alone describe possible unknown faults in the tool developed. The validation method conducts software test cases that either pass or fail. Resulting failures are then addressed individually, or reclassified in the difference taxonomy. Where an explanation is provided for a failure, the discussion must be viewed as speculative because all concrete differences identified have been integrated with the analysis taxonomy.

One design goal for the module developed is to provide coverage for a breadth of OS versions and kernel architectures. These test cases are intended to demonstrate that coverage by representing both i386 and x86_64 Intel architectures over the span of minor OS X versions (10.6.0-8 and 10.7.0-3) within the current and previous releases of the operating system. All tests are performed on guest installations of OS X running as a VM. This setup offers the linear file format volafox requires in analyzing 64-bit kernel memory, the contents of which are written to disk when the VM is suspended. Efforts were made to minimize OS interference with the state of open files during collection. Specific modifications include: removing network interfaces, deleting startup items, and disabling the OS X automatic file indexing process known as Spotlight. The sole installation of OS X Server also had the `servermgrd` daemon disabled to eliminate its numerous child processes on startup.

Configurations for the controlled test cases include:

1. OS X version: 10.6.8
   Darwin kernel architecture: i386
   RAM installed: 1 GB

2. OS X version: 10.6.0 Server
   Darwin kernel architecture: x86_64
   RAM installed: 1GB

3. OS X version: 10.7.3
   Darwin kernel architecture: i386
   RAM installed: 2GB

4. OS X version: 10.7.0
   Darwin kernel architecture: x86_64
   RAM installed: 2GB

Table 5 summarizes results across the four controlled test cases. After accounting for constraints, deficiencies, and explained differences listed in the analysis taxonomy (not shown), this table indicates how similar the volafox open files output is to the `lsof` approximation. Failures in the comparison are marked in red and discussed in order from top to bottom of the table.

The extra process in the volafox output (F2) for the 10.7.x cases is a daemon with the highest PID in the process list. It therefore appears to have been launched after executing `lsof`, explaining its absence in the baseline listing in both instances.

While the username deficiency (D1) is not classified as a failure, it is listed in the table to emphasize the number of handles affected by this bug.

The additional volafox file descriptors (F3) in the 10.7.3 test case, and three of the four in the 10.7.0 case belong to `launchd`. Because the `launchd` process manages all other daemons (Singh 2006, 38), it is very active and therefore volatile. For both 10.7.x test cases the `lsof` and `launchd` processes appear to be confounded, though similar problems were not observed in the 10.6.x test cases. These differences are believed to represent normal OS interference with the state of open files between the time `lsof` is executed and the VM is frozen.

Table 5. Difference summary for controlled test cases.

Diff	Field	10.6.8 i386	10.6.0 Sever x86_64	10.7.3 i386	10.7.0 x86_64
F1	COMMAND	0	0	0	0
F2	PID	0	0	+1	+1
D1	USER	15%	17%	38%	19%
F3	FD	0	0	+3	+4
F4	mode	0	0	0	0
F5	TYPE	0	0	0	0
F6	DEVICE	0	0	0	0
F7	SIZE/OFF	1	1	1	1
F8	NODE	2	1	1	1
F9	NAME	0	0	0	0

The fourth extra file descriptor (F3) in the 10.7.0 test case appears to be a malformed vnode. All members within the structure are invalid, and the file name is made up of non-ASCII characters. This case does call into question the methodology described in Section 3.2 for determining valid descriptors in the file table. Since the occurrence appears to be isolated, it is particularly difficult to debug this potential implementation failure. One possible explanation is that the handle may be an initialized but as-yet-unused vnode in the file descriptor table. Luckily, the error output is well-handled and therefore a human analyst should be able to make this determination with ease even if the tool cannot.

In all four test cases, the file size failure (F7) is for the pseudo-tty device opened by process `Terminal`. The Terminal application is in the process hierarchy for `lsof`, which as explained in E7 is known to modify some `ttys` device offsets during execution. This explanation might have led to another explained difference in the taxonomy, but detection could not be easily automated for this case.

In all four test cases, the node identification failures (F8) belong files related to time zone opened by the `notifyd` process. It is unclear why the notification server makes changes to these files during `lsof` execution and additional knowledge of OS X internals is needed to analyze this failure further. However, because the difference in node value is always observed on regular files but only those associated with time zone and this particular process, it is not believed to be an implementation fault.

Results from the four controlled test cases yield several important conclusions. First, the volafox open files module is functional for kernels utilizing both Intel i386 and x86_64 architectures. Second, the tool provides coverage for the OS X 10.6.x Snow Leopard and 10.7.x Lion operating systems. Third, the username deficiency (D1) results suggest that this field cannot be trusted in the volafox output. Finally, the low number of unexplained failures suggests the implementation is successful under the research definition.

4.3. Real-world Data Analysis

In addition to the controlled test cases, the volafox handles module was also tested against a set of memory collected from physical machines. The script `capture.py` was developed to automate collection of memory using the MMR tool and a variety of incident response data, including `lsof`, for comparison. These real-world collections are invaluable for

program debugging and revealing edge cases in the handles implementation but are not well suited for validation for several reasons. First, because failures cannot be replicated it is difficult to determine if a fault is caused by implementation bug or validation accuracy. Second, the collection time required by MMR assures that output from `lsof` is always stale when compared to the memory capture. Finally, the real-world data available does not cover the breadth of OS versions and kernel architectures.

Revision 52 of the volafox project does not support the MMR output format directly. As a result, only i386 captures are analyzed with volafox after conversion to linear format using the `flatten.py` utility. Ten qualifying samples were collected from real Mac computers, eight of these running 10.6.8, one 10.7.0, and one 10.7.2.

Table 6 shows a combined summary of the real-world results. Because the hardware and software configurations vary greatly between collections, the data points represent different sample populations that cannot be aggregated to produce valid mean or standard deviation. Instead, the range of each constraint, deficiency, explained difference, and failure is reported to offer a general impression of how commonly these differences occur. A few noteworthy conclusions emerge from this analysis.

1. With up to 10% of processes (F2) and 22% of handles (F3) thrown out for comparison during alignment, `lsof` does *not* approximate the real-world data very closely.

2. The set of non-vnode handles (sockets, pipes, semaphores, etc.) make up a significant portion of the `lsof` results (C2). Sockets in particular are of high investigative value and should therefore be considered in future work.

3. Unsupported file systems (C3) in the real-world data were cross-referenced with the mount information also collected by the `capture.py` script to determine which types should be considered for future support. The results included one instance each of: `msdos` (FAT32 external hard drive), `cddafs` (responsible for reading audio CDs), `ntfs` (Apple Bootcamp installation of Windows), and `mtmfs` (used to implement the Mobile Time Machine feature).

4. Explained differences (E1-E7) and the `/dev` sizing deficiency (D2) do not affect a large number of processes and handles. However, their enumeration is important because it filters the number of failures that must be considered.

5. For a given handle the size/offset (F7) and node identifier (F8) information can be particularly volatile, with up to 10 and 8 percent change observed respectively.

Table 6. Combined real-world results (10 samples).

Diff	Description	Quantity or % Per Sample
C1	SOCKET handles cannot be subtyped	15-22% of handles affected
C2	Non-vnode handles are not fully supported	27-40% of handles affected
C3	Non-HFS+/DEVFS vnodes are not fully supported	0-4% of handles affected
D1	Δ USER field	16-54% of usernames misreported
D2	`/dev` directory cannot be sized	0-1 handles affected
E1	`lsof` process is not shared	0-1 process removed
E2	`MacMemoryReader` and `image` processes are not shared	0*-2 processes removed
E3	`sh` process is not shared	0*-1 process removed
E4	Duplicate handles labeled FD: '`twd`'	2-5 handles removed
E5	LINK handles are mislabeled	0-3 handles affected
E6	FIFO handles do not report device identifier	0-2 handles affected
E7	`lsof ttys` file size is not shared	0*-13 handles affected
F1	Δ COMMAND field	0 commands differ
F2	Δ PID field	0-10% of processes removed
F3	Δ FD field	4-22% of handles removed
F4	Δ MODE field	0-2 modes differ
F5	Δ TYPE field	0-2 types differ
F6	Δ DEVICE field	0-2 device identifiers differ
F7	Δ SIZE/OFF field	0-10% of sizes/offsets differ
F8	Δ NODE field	0-8% of node identifiers differ
F9	Δ NAME field	0-3% of names differ

6. Upon manual inspection of the failures, high volatility of the name field (F9) was often linked to two applications: Spotlight and the Microsoft suite. Spotlight is Apple's indexed search technology and automatically begins processing external media when mounted. Because the `capture.py` script is delivered on external media, the act of collection increases indexing activity.

The real-world data identifies a number of implementation problems that may not have been encountered otherwise. For example, the E2, E3, and E7 results include an asterisk because one of the samples experienced an interesting collection failure. The `capture.py` script and all its associated processes (`Python`, `sh`, `MacMemoryReader`, `image`, etc.) are all absent from the volafox output for this sample, making it clear the processes list had been truncated. Due to the high occurrence of invalid pointers observed in the real-world data and several volafox execution errors, additional exception handling was added to the `lsof` module to support debugging. The new code identified several cases where the underlying linked data structures were broken in the memory image. In a real investigation these occurrences might represent evidence lost. One recommendation to mitigate this problem is to assure memory capture

proceeds as rapidly as possible. One factor found to affect capture speed in real-world collections is the type of external media used to store the image. Timing results recorded by `capture.py` showed a 16 Mb/s average increase in capture speed when using an external hard drive over flash storage.

5. Conclusions and Future Work

This paper presents documentation and implementation of a new capability for parsing file handles from an OS X memory capture. Initial development of the module required performing a manual design recovery of the data structures responsible for handling files for OS X. To alleviate the manual recovery process in future versions of OS X, a novel header-processing tool programmatically parses structures defined for different kernel architecture and OS versions and converts these into templates used by the file handle module.

Testing the implementation identified several areas for future work not directly related to the research goal. First, the open files module does not reliably output the correct user of a running system process. No fault could be identified in the implementation, nor any problem with the kernel structure analysis described in prior work. Second, memory captured on physical hardware suffers from a high number of invalid pointer references, occasionally resulting in malformed linked-lists. Robust exception handling needs to be implemented throughout volafox to address this problem in a memory analysis tool.

Finally, several additional modules must be developed to establish volafox as an analysis tool suited for technical users and forensic examiners. The National Institute of Standards and Technology (NIST 2006) describes the minimum requirements for volatile collection during incident response. At present, the volafox tool includes modules for parsing several of these requirements but is still missing a list of login session, network configuration, and operating system time.

Acknowledgements

This research was supported by the U.S Air Force Cyberspace Technical Center of Excellence and the U.S. Air Force Office of Scientific Research (AFOSR/RSL). Views expressed in this paper are those of the authors and do not reflect the official policy or positions of the U.S. Air Force, U.S Department of Defense, or the U.S. Government.

Andrew Hay received a M.S. from the Air Force Institute of Technology and completed a thesis based on the research in this paper. He has a B.S. in Computer Science from the University of Arizona. Research interests include mobile device forensics, memory analysis, and network security.

Gilbert L. Peterson is an Associate Professor of Computer Science at the Air Force Institute of Technology, and Vice-Chair of the IFIP Working Group 11.9 Digital Forensics. Dr. Peterson received a BS degree in Architecture, and an MS and Ph.D in Computer Science at the University of Texas at Arlington. He teaches and conducts research in digital forensics, and statistical machine learning.