FORENSIC MEMORY ANALYSIS FOR APPLE OS X

THESIS

Andrew F. Hay

AFIT/GCO/ENG/12-17

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCO/ENG/12-17

FORENSIC MEMORY ANALYSIS FOR APPLE OS X

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Andrew F. Hay, BS

June 2012

AFIT/GCO/ENG/12-17

FORENSIC MEMORY ANALYSIS FOR APPLE OS X

Andrew F. Hay, BS

Approved:

Gilbert L. Peterson, PhD (Chairman)	21 MAY 2012 Date
Barry E. Mullins, PhD (Member)	21 May 12 Date
Jonathan W. Butts, Maj, USAF (Member)	21 May 2012 Date

Abstract

Analysis of raw memory dumps has become a critical capability in digital forensics because it gives insight into the state of a system that cannot be fully represented through traditional disk analysis. Interest in memory forensics has grown steadily in recent years, with a focus on the Microsoft Windows operating systems. However, similar capabilities for Linux and Apple OS X have lagged by comparison. The volafox open source project has begun work on structured memory analysis for OS X. The tool currently supports a limited set of kernel structures to parse hardware information, system build number, process listing, loaded kernel modules, syscall table, and socket connections. This research addresses one memory analysis deficiency on OS X by introducing a new volafox module for parsing file handles. When open files are mapped to a process, an examiner can learn which resources the process is accessing on disk. This listing is useful for determining what information may have been the target for exfilitration or modification on a compromised system. Comparing output of the developed module and the UNIX `lsof` (list open files) command on two version of OS X and two kernel architectures validates the methodology used to extract file handle information.

Acknowledgments

I would like to thank my research advisor, Dr. Gilbert Peterson, for sharing his extensive knowledge throughout this process and always freely offering advice when I needed it most. Without his support for my eccentric research interests this work would not have been possible.

Andrew F. Hay

Table of Contents

Page

List of Figures

Page

List of Tables

Page

FORENSIC MEMORY ANALYSIS FOR APPLE OS X

I. Introduction

Desktop and mobile computing platforms are central to many law enforcement and enterprise investigations due to their enormous capacity for digital evidence. Identification, individualization, association, and reconstruction of such evidence are all steps of the *forensic process* (Inman & Rudin, 2002). Professionals rely on both technical expertise and appropriate tools when applying these steps to digital sources. Because an examiner cannot always anticipate the platforms encountered during the course of an investigation, he or she must be prepared to deal with any of them. Tools for data acquisition and analysis are therefore required across a breadth of common endpoint devices in order to assure needed flexibility, Apple's OS X among them.

Apple currently holds the number three position in PC manufacturing (Schonfeld, 2012), with its OS X operating system representing about 10% domestic market share (Donnini, 2012) and 6% globally (Net Applications, 2012). While this accounts for a small minority when compared with Microsoft Windows, OS X has become the operating system of preference for many individuals. As a result, it cannot be ignored as a possible target during forensic investigation.

Forensic memory analysis has become a critical capability in digital forensics because it allows insight into the state of a system that cannot be fully represented through traditional disk analysis. From an investigative standpoint, computer memory may be the only source of data capable of revealing the use or misuse of computer at time

of seizure. Additionally, it may hold the sole evidence of malware or other compromise influencing the conclusions of the examiner. Interest in memory forensics has grown steadily in recent years, with a focus on the Microsoft Windows operating systems (Cohen & Collett, 2008). However, similar capabilities for platforms such as OS X have lagged by comparison.

Memory analysis depends on capabilities for RAM acquisition and tools for abstracting information from raw memory. Though tools exist to capture memory on OS X (Section 2.4), the sole project focusing on structured data analysis (Lee, 2011) is immature by comparison to its counterpart for Windows and Linux (Volatile Systems, 2012). This leaves primarily context-free options available to the forensic examiner, such as string searches and file carving. The challenge is to perform better than context-free memory analysis so ultimately tools can be developed to replace more invasive incident response methods for determining the state of a running system. This research addresses one component in the suite of data collected during live response: a list of file handles associated with running processes.

When open files are mapped to a process, an examiner can learn 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. Handles may also help identify a suspicious process when unexpected file access or modifications are observed. For example, malware masquerading as an innocuous executable while editing log files to its cover tracks. Because open files can help characterize process activity and highlight misuse of a computer during an investigation, it is desirable to recover this information from a memory capture.

1.1 Research Objectives and Assumptions

The goal of this research is to implement and document a capability for parsing file handles from raw memory captured on OS X. Two objectives are derived to support this goal. First, perform design recovery of the kernel data structures responsible for handling open files. Second, develop a flexible process for programmatically handling structures defined for different kernel architectures and operating system versions. This necessitates extensible software design resilient to changes in future versions of OS X.

Assumptions related to achieving the research goal include:

1. A method already exists for copying physical memory to file in a forensically sound manner.
2. Because analysis is always performed on a forensic duplicate of imaged memory, the tool developed is not guaranteed nor required to ensure data integrity.
3. Prior work can determine the addresses associated with key kernel symbols and perform virtual to physical address translation.
4. Direct validation of tool output is not possible and must therefore be compared with approximate data, the acquisition of which alters the state of memory.
5. Code defining the parsed kernel data structures is open source.
6. The set of architectures under consideration is limited to Intel i386 and x86_64.
7. The set of OS X operating systems under consideration is limited to 10.6.x Snow Leopard and 10.7.x Lion.
8. The set of handle types supported by the implementation is not comprehensive.

The first two assumptions leverage existing work described in Sections 2.4 and 2.5.2 respectively. Validation challenges are addressed in Section 4.1. Design assumptions and emergent research limitations are discussed throughout Chapter 3 and formally classified in Section 4.1.2.

1.2 Methodological Approach

The open source volafox project (Lee, 2011) has begun the work of analyzing raw memory dumped on OS X with modules for parsing hardware information, system build number, process listing, loaded kernel modules, syscall table, and socket connections. One omission in the existing volafox feature set is an ability to list file handles. This research presents the memory analysis methodology required to extract file handles in use by each process, and demonstrates its application through a new volafox module. The research module outputs handle information corresponding to process name, PID, file descriptor, access mode, handle type, size/offset, catalog node ID, and name. Supported handles include regular files, directories, symbolic links, character special files, memory-mapped files associated with the process executable and its current working directory.

Chapter 3 describes methodology for implementing the new file handles module. Software design requirements and constraints are listed along with a format specification for the information displayed to the user. Next, design recovery of key kernel data structures is presented. The relevant members and relationships of 36 xnu kernel structures are used to parse the required handle information. Package organization, key objects, and functions from the existing volafox source code are then described. Object-oriented design of the new file handles module is depicted using UML and a description of the data structure template development process is presented. Chapter 3 closes with a discussion of outstanding implementation issues. Results of the new module are validated against the UNIX command-line tool `lsof` (list open files) in Chapter 4. Versions 10.6 Snow Leopard and 10.7 Lion of the OS X operating system for both 32 and 64-bit kernels are the tested configurations.

1.3 Research Implications

Design recovery of the OS X kernel data structures related to process file handles in Section 3.2 and summarized in Appendix B is a primary implication of the research. Development of the new volafox module for parsing handle information exists in part to demonstrate this part of the methodology in a practical form. While core UNIX technologies lead to many similarities with other systems, OS X's unique design architecture makes it distinct from other platforms. The research is novel because kernel structures parsed by the tool differ from those considered in the literature for Linux and Windows.

The research module also demonstrates a process developed for dynamically handling data structure layout in memory across multiple architecture and operating system versions. The process is broken down in three parts. Section 3.4.2 first describes how C struct templates are built to an interface specification for each OS and architecture configuration. Next, a solution is given for selecting the correct template at runtime using an abstract object initializer. Finally, Section 3.4.3 presents an external C program for generating template syntax using the kernel header files defining key structures. The process itself adds value because it simplifies program logic and readability of the resulting source code. Though unverified, the process is extensible by design and therefore may also support future versions of OS X and additional kernel structures outside the immediate research scope.

Finally, the new volafox module for listing open files is validated in Chapter 4 and found to reasonably approximate output from the UNIX `lsof` command. After addressing outstanding constraints and deficiencies, work based on this tool could one

day replace at least one required executable from an incident response toolkit. This result directly addresses the problem statement and therefore implies a successful research outcome.

II. Literature Review

Digital evidence is integral to modern forensic investigation because of its pervasiveness in the consumer, enterprise, and government domains. Practitioners require tools to simplify the collection and analysis of such evidence across the breadth of endpoint devices encountered in the field. An investigator may employ tools from a variety of open-source, commercial, and underground sources in pursuit of needed capabilities. The work presented in this thesis evolves an existing open-source project for analyzing the memory of Apple's OS X desktop operating system.

This chapter introduces digital forensics and the related process of incident response, with specific focus on memory analysis. Strategies for capturing memory on OS X are discussed as a component of incident response, and necessary precursor to analysis. A review of existing work for analyzing raw memory drumped on Linux serves as a baseline for higher-abstraction analytic capabilities. Finally, the volafox tool being extended for this research is presented, along with its methodology for acquiring kernel symbols and parsing key data structures.

2.1 Digital Forensics

The National Institute of Justice (NIJ) describes *digital evidence* as "[i]nformation stored in binary form that may be introduced and relied on in court" (2008, p. 52). In order to meet the latter half of this definition, evidence must be handled in a manner

consistent with accepted forensic practice. Generally accepted standards include at minimum (NIJ, 2008):

- A process for ensuring evidence remains unchanged.
- Assuring technical expertise of personnel responsible for analysis.
- Detailed documentation of all actions affecting the evidence.

A similar definition by the National Institute of Standards and Technology (NIST) describes *computer forensics* as the "practice of gathering, retaining, and analyzing computer-related data for investigative purposes in a manner that maintains the integrity of the data" (2008, p. D-1). *Digital forensics* is therefore a process characterized by the preparation for, and execution of, the following activities (NIST, 2006):

1. *Data collection* – identifying and preserving useful data from all possible sources of digital evidence according to a standardized subprocess.
2. *Examination and analysis* – a methodical approach to correlating evidence from various sources to determine the characteristics and impact of an event.
3. *Reporting* – a procedural log and analysis summary used to demonstrate the integrity of the process overall and present its findings.

Given the wide array of platforms and environments capable of supporting digital evidence, these definitions describe *what* the digital forensic process requires rather than *how* to accomplish it.

The implementation described in Chapter 3 is specific to the evidence analysis component of the three-part process above. However, in order to understand the input for analysis, the collection of digital evidence is discussed here.

2.2 Incident Response

Motivation for applying the digital forensic process is known as an *incident*, "violation or imminent threat of violation of computer security policies, acceptable use policies, or standard security practices" (NIST, 2008, p. D2). Organizations with a mission involving digital forensics need an established policy for handling such events that describes recommended practices for mitigating the effects. Performing the actions prescribed in such a policy is defined as *incident response*, an organizational standard describing *how* the process of digital forensics is to be applied within a specific context. An incident response almost always incorporates components of data collection and reporting as specified by the digital forensics process. Depending on mission needs of the affected systems, analysis may take place on site, be deferred, or a combination of the two. Deferred analysis may be necessary when it requires technical skills beyond that of incident responder, tools unavailable at the scene, or is expected to take more time than can be reasonably allocated.

The tool implemented in Chapter 3 could operate as a component of incident response but is more likely to be employed during deferred analysis. Data used as input to the tool is volatile and therefore its capture must be conducted as part of an incident response policy in order to make analysis possible.

2.2.1 Volatile Data.

Volatile data describes digital evidence of a dynamic or transient nature that is sensitive to time, system state, or power. It is contrasted with nonvolatile data such as that stored on a magnetic hard drive, smart card, or flash memory (NIJ, 2008). While sources of nonvolatile data may be collected directly as hardware or imaged during incident

response, volatile data is fragile and therefore has distinct collection and analysis needs (NIST, 2006). This section motivates the collection of volatile data during incident response as a compliment to traditional disk analysis, and provides an overview of the process. Note that system characteristics and the collection process described focus on Linux systems due to similarity with OS X.

Several arguments for seizing and analyzing volatile data exist. First, no usage snapshot is complete without a record of what the machine was doing before shutdown. This action is usually a prerequisite to imaging a system for deferred disk analysis (NIST, 2006). Valuable evidence such as running processes, network connections, and currently open files may all be irretrievably lost as soon as the target is powered off (NIST, 2008). Critical systems may also be unavailable for imaging because the downtime would adversely affect user experience or services, making traditional disk analysis infeasible. The so-called *Trojan Defense* is also a concern. One classic example is Aaron Caffrey's acquittal following a defense which could have been disproven using evidence in volatile data (Leyden, 2003). Modern malware is specifically engineered to avoid the disk as a means of preventing detection. This means volatile data may be the only source of evidence available to prove its presence or absence. Dolan-Gavitt (2008a) demonstrates how a cached version of the Windows registry can be exploited without leaving any evidence on the disk.

The slow but progressive adoption of full-disk encryption offers what may be the single best argument for integrating volatile data collection into any reasonable incident response (Casey, Fellows, Geiger, & Stellatos, 2011). Data on disk may be essentially worthless to the investigator once the target is powered off if its contents become fully

encrypted. Full disk encryption became a standard feature on OS X when FileVault 2 was included with the release of 10.7 Lion. Tools for breaking FileVault 2 encryption via memory exploit are now available both open source and commercially (Kessler, 2012; Maartmann-Moe, 2012). However, none of the known attacks work after the target is turned off.

2.2.2 Volatile Collection Process.

Recovering volatile data from a Linux target typically involves a toolkit consisting of trusted executables needed to dump specific operating system information, exfiltrate output, and maintain integrity of evidence (Mandia, Prosise, & Pepe, 2003). On UNIX systems and their derivatives, the toolkit can be built using commands available on the platform, compiled as static binaries verified using the `ldd` tool on Linux (Burdach, 2004), or `otool` on the Mac (Webb, 2010). The volatile data of interest and some of the Linux commands associated with its collection are summarized below (Burdach, 2004; Carvey, 2009; Mandia et al., 2003):

- `date` – system date and time
- `w` – login sessions
- `ls` – directory listing including MAC times
- `netstat` – network connections and open ports with associated applications
- `ps` – process list
- `arp`, `route` – arp and routing cache tables
- `cat /proc/modules` – loaded kernel modules
- `lsof` – process-to-file mappings
- `cat /proc/version` – OS version
- `cat /proc/sys/kernel/name` – host name
- `cat /proc/sys/kernel/domainame` – domain name
- `cat /proc/cpuinfo` – hardware info
- `cat /proc/swaps` – swap partitions
- `cat /proc/partitions` – local file systems
- `cat /proc/self/mounts` – mounted file systems

- `cat /proc/uptime` – uptime
- Physical memory image (kernel-specific)
- Clipboard contents (distribution-specific)
- Command-line history (shell-specific)

The toolkit is typically delivered to the target via read-only media to protect its integrity, and its output usually stored to external media or piped over a network connection. After constructing such a toolkit, commands for collecting, naming, and storing the evidence collected are scripted for the native command-line interface. The command order should give consideration to the perishability of the data (NIST, 2006):

1. Network connections
2. Login sessions
3. Contents of memory
4. Running processes
5. Open files
6. Network configuration
7. Operating system time

Webb offers details on toolkit compilation and scripting for incident response on OS X which roughly approximates the aforementioned process (2010).

A possible alternative to this incident response methodology by Choi, Savoldi, Gubian, and Lee (2008) integrates both collection and analysis. The Linux Evidence Collection Tool (LECT) is a framework for live evidence collection aimed at server investigations. It consists of a console-based collection tool and graphical analysis environment. The tool emphasizes collection of log files and other disk-borne data for offline analysis and correlation with volatile data. Because of this hybrid approach, LECT is not well aligned with the research goals of this thesis.

The preceding exploration of volatile collection strategies for Linux serves to characterize the kinds of information that are most critical to the investigator. In the

following section, an alternative methodology is proposed to replace many of the commands mentioned with a tool for parsing this information from an image of physical memory. Use of a volatile collection toolkit as described in this section establishes a baseline against which memory forensics can be measured in terms of both analytic power and system impact.

2.3 Memory Forensics

This section describes background needed to understand the sub-field of digital forensics concerned with analyzing the contents of raw memory. Random-access memory (RAM), physical memory, or main memory are all terms used interchangeably to describe the first backing store for the operating system's virtual memory manager (Gorman, 2004). Physical memory is of interest to the forensic examiner because it stores current and recent data being acted on by the CPU and various memory-mapped I/O. Suiche (2010) asserts that on UNIX systems the term *physical memory* is synonymous with `/dev/mem`, the character device used to abstract the hardware implementation.

A variety of hardware and software solutions exist to copy the contents of physical memory to file for offline analysis. Output of such a tool is commonly referred to as an *image* or memory dump. The sophistication of memory analysis varies, but by one definition represents an "attempt to use memory management structures in computers as maps to extract files and executables resident in a computer's physical memory" (Urrea, 2006, p. 2). The more specific definition required by this research classifies *structured memory analysis* as the examination of kernel structures present in an image to partially characterize the state of a computer at time of capture. The two-part process of

analyzing physical memory generally involves the capture of RAM during incident response in a manner consistent with forensic principles, and subsequent examination of its contents using tools capable of parsing kernel structures or other useful patterns from the resulting file.

In its simplest form, the value of memory introspection will be familiar to any programmer who has used a debugger to analyze a core dump. Exploit developers use many of the same tools for analyzing the state of RAM during execution to craft payloads for corrupting memory in order to achieve a desired effect (Miller & Zovi, 2009). Analysts on both sides of the ethical divide observe memory in action to reverse engineer systems, applications, or malware. From a forensic standpoint, the state of a system can be most comprehensively recorded by capturing the raw contents of RAM. However, modern computers guarantee this information alone is incomplete and fragmented due to virtual memory and other architectural mechanisms. Further complicating its analysis are operating system protection schemes such as Address Space Layout Randomization (ASLR), which seek to obfuscate the location and structure of memory as a means of safeguarding information.

2.3.1 Advantages of Memory Analysis.

Despite the challenges, there are several compelling reasons to prefer structured memory analysis over the techniques for investigating volatile data described in Section 2.2.1. First, because memory contains both in-use and recently-used data, it is possible to reconstruct a limited timeline of system usage. Recently used memory segments may provide information not represented in the log files and other incident response toolkit output because, like free disk space, these segments are marked for recycling by the

operating system. Solomon, Huebner, Bem, and Szezynska assert that "[d]espite the fast decay of user pages in free memory, it is still a worthwhile source of forensic data" (2007, p. 71). The most recent work on Linux even suggests that it may be possible to go about such reconstruction in a coherent and organized fashion (Case, Marziale, Neckar, & Richard, 2010).

A second advantage of structured memory analysis is its resilience to system tampering when compared with other strategies for reconstruction using disk analysis or incident response toolkit output. Syscall hooking, log manipulation, and other anti-forensic measures are common features of modern malware (Ligh, Adair, Hartstein, & Richard, 2011; SANS Institute, 2008). These exploitation tactics can lead to missed evidence and incorrect conclusions if the results of other analysis are not compared against evidence in memory (Hay & Nance, 2009). Even when employing static executables, memory analysis may be the only way to reveal running malware when faced with a sophisticated kernel rootkit. Dolan-Gavitt (2008a) also demonstrates how physical memory may contain the only evidence of registry tampering on Windows.

The third benefit is that physical memory may contain evidence that is never saved to disk and would otherwise be lost. Consider that one reason for performing hard power-off of a system prior to imaging is to preserve the temporary files that might be overwritten in a graceful shutdown operation (NIST, 2006). This action also results in the unfortunate loss of any files that have not been saved because they exist only in memory at the time of shutdown. If memory is imaged during the incident response, these files are preserved and "sections of memory can have more traditional forensic procedures applied

to them such as file carving and hashing" (Case, Cristina, Marziale, G. G. Richard, & Roussev, 2008, p. S70).

Finally, volatile data collection using a scripted response toolkit has an inherent disadvantage: changes to the system state which occur as a result of collection. Dumping information in a forensically sound manner means "[s]ome of the response tools may even substantially alter the digital environment of the original system" (Law, Chow, Kwan, & Lai, 2007, p. 137). A trusted tool must overwrite existing memory on the target system when both the executable and all its static libraries are copied from disk in order to preserve the forensic integrity of the output. Next, a series of I/O operations are needed to support the exfiltration of the output. The process is then repeated for each distinct piece of volatile data sought. This leads to a situation where "[t]he credibility of the acquired data relies solely on the reliability of the tool and the expertise of the user" (Law et al., 2009, p. 1). By contrast, capturing physical memory for deferred analysis has the potential to be much less invasive. Some techniques, such as FireWire acquisition, do not require any execution on the target system. Similarly, suspending a virtual machine allows memory capture on the guest with minimal impact.

The lack of investigator-friendly methods for analyzing memory images provides the facing argument to the use of structured memory analysis outlined in the preceding paragraphs. When deciding between an imperfect incident response toolkit and an indecipherable collection of bits, it is clear that a potentially corrupt dataset is preferable to no information at all. A dump of raw memory is only as useful as the tools available to abstract information for human analysis. Therefore the acquisition and analysis capabilities for this method are equally important.

2.3.2 Memory Forensics Process.

Analysis of raw memory is procedurally similar to many types of digital evidence, generally following the steps of capture, analysis, and reporting. Carvey (2009) provides a description of the memory forensics process, with a focus on Windows since it is the only platform where this type of analysis has matured.

The first step is to choose a method of capture based on the hardware, operating system, state of the target, and context of the investigation. Next, a delivery method is selected. Most common today is a USB device that acts as both capture toolkit and storage for the resulting memory image. The toolkit hash is documented prior to use, and appropriate physical and policy measures taken to prevent altering the contents of the device before incident response. When the tool is executed on a target, system time and hash value of the output are documented in the incident response log. Carvey (2009) argues the hash should be recorded once memory dump is complete due to changes that occur during the capture process. After memory is captured, measures must again be taken to avoid changes to the storage media. Analysis occurs only on copies of the original image file, the integrity of which can be verified using the recorded hash. This is done using a write-blocker on a forensic workstation capable of copying the storage media without changing it. Analysis then proceeds on the copy according to the needs of the investigation and the tools available. The process for parsing any results is documented in detail such that it can be reproduced, and the conclusions of the examiner are summarized in a written report. Since the reporting aspect is largely determined by organization policy or legal requirements it is not discussed further, but the details of capture and analysis for OS X and similar platforms are now presented.

2.4 Mac Memory Acquisition

This section focuses on physical memory capture for the platform under research consideration: OS X running on Intel architecture. While this thesis addresses primarily the analysis component of the memory forensics process, acquisition of RAM is a prerequisite to making such research worthwhile and is therefore discussed first.

2.4.1 FireWire, Cold-boot, and VM Extraction.

The IEEE 1394 or FireWire interface is susceptible to direct memory access (DMA) on OS X as well as Windows. Its vulnerability, the underlying PCIe bus, is shared by a variety of I/O on the Mac including ExpressCard, SD slot, and the new Intel Thunderbolt port (Graham, 2011). Using this interface to reliably capture RAM on the Mac was first demonstrated by the `pyfw` attack (Becher, Dornseif, & Klein, 2005). While useful, the project is not explicitly designed as a forensic tool (Hermann, 2008). Numerous forensic implementations have emerged since, including Goldfish (Gladyshev & Almansoori, 2010, 2011), and `libforensic1394` (Witherden, 2010).

There are several shortcomings to DMA capture. First, the vulnerability is limited to the first 4GB of system memory (Gladyshev & Almansoori, 2010). Kubasiak and Morrissey further characterize the method as "somewhat invasive and involves tricking the system into thinking an iPod is being connected" (2009, p. 528). Finally, the attack is easily mitigated using the lock screen if user switching is disabled (Garrison, 2011).

The alternative Kubasiak and Morrissey suggest to FireWire exploitation is `msramdump` (2009, p. 528; McGrew, 2008). This technique calls for cold boot extraction, where the target system is forcibly shutdown and then quickly booted from external media before the contents in memory are fully wiped by momentary loss of

power. This tool builds on earlier research where the hardware DIMMS were physically cooled and removed from the target so the contents could be read externally. The risk of evidence loss appears to be high with this strategy and the authors suggest it only as a method of last resort for extracting the decryption key of a Mac using FileVault.

Another option is to copy the memory backup file maintained by the host system of an OS X virtual machine (VM) during suspension. Ligh, Adair, Hartstein and Richard list the location of such files for various virtualization products and describe how they can be used with Volatility for Windows analysis (2011). This represents perhaps the least invasive method available, however there are two caveats. First, until the release of OS X 10.7 Apple's software license agreement only permitted virtualization of its server operating system. Second, 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 examiner. Possible exceptions include enterprise installations of OS X Server or VMs used for research, software development, or reverse code engineering.

2.4.2 Kernel Module Capture.

Classic methods for extracting memory from UNIX systems involved reading directly from the `/dev/mem` or `/dev/kmem` character device using `dd`. This ability has been disabled in the Linux kernel for some time and when OS X transitioned to Intel architecture it became similarly depreciated on the Mac. Singh (2006a) describes the problem and makes several suggestions for implementing a custom version of `/dev/kmem` to read from kernel memory directly. This approach was demonstrated by Suiche (2010) with an emulation of `/dev/mem` used to dump RAM on a Mac. The

presentation also describes how critical symbols retrieved from the `mach_kernel` executable can be used to build a kernel memory manager capable of virtual to physical address translation. Such translation is required to fully browse kernel address space and copy its contents. This capability did not become publically available until the release of Mac Memory Reader (Architecture Technology Corporation [ATC], 2011), a free component of the commercial Mac Marshal product. Technical limitations and design decisions lead to several details worth nothing about the tool's output (ATC, 2011; Inoue, Adelstein, & Joyce, 2011; Leat, 2011):

- Output file format is a Mach-O, equivalent to Linux ELF (Apple Inc., 2009).
- The `-H` option prints MD5, SHA-1, SHA-256, or SHA-512 hash to `stderr`.
- Using '`-`' for the output filename prints to `stdout` for command piping.
- Physical memory map used is the same format as the `showbootermemory` macro in the Apple Kernel Debug Kit.
- Only addressable pages can be copied from physical memory, excluding a small number of pages missing from the memory map.
- Memory segmentation is maintained along with offsets in the file using a header listing, the results of which can be interpreted with Apple's `otool`.
- Memory-mapped I/O device segments and memory ports are not captured.
- Memory allocated to a virtual machine hypervisor is not captured.

There are several disadvantages to this form of acquisition. First, Mac Memory Reader requires administrator privileges to load the needed kernel module. While this may limit its use during some investigations, in many cases a cooperative administrator may be available to provide the password. Second, output from such a tool could be corrupted by the presence of memory forensic countermeasures (Haruyama & Suzuki,

2012) 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” (Case et al., 2008, p. 2). Finally, because a kernel module must be loaded into the memory in order to perform the capture, its use alters the target system. The problem is addressed by the Mac Memory Reader README text (ATC, 2011):

> Pieces of the MacMemoryReader executable code and data will certainly appear within the RAM snapshot, simply because MacMemoryReader is running in the same memory space being acquired. This is a known "footprint" and aspect of live analysis.

While this represents a violation of the first forensic principle outlined in section 2.1, it is likely to be no worse than the other practical methods discussed (VM analysis notwithstanding). This method of capture also has the advantage of being self-documenting, in that its usage is represented within the resulting output. Despite the challenges, availability of this robust acquisition capability for the Mac encourages additional research and emphasis on analytic capabilities for the platform.

2.5 Structured Memory Analysis

Most literature and web references to memory analysis on OS X discuss context-free techniques such as string searches, manual hex examination, file carving and the like. Valenzuela (2011) writes about these rudimentary methods in a blog post responding to the release of Mac Memory Reader. Malard (2011) expands these approaches to include session information and password extraction in a paper discussing hacking tactics for OS X. One early effort to achieve systematic analysis of raw memory on the Mac is a tool for

automatically extracting login credentials using the FireWire exploit cited previously (Makinen, 2008). The attack iterates over all pages in memory searching for a string flag known to occur at a particular offset and then performs a keyword search for 'username' and 'password' within that page to reveal the target information. This tool is valuable for decrypting disks with FileVault enabled but is only effective against OS X 10.4 Tiger. Unfortunately, all context-free methods for memory analysis are inherently limited, imprecise, and inefficient.

2.5.1 Linux Memory Analysis.

This section offers a review of existing work concerning higher-abstraction memory analysis, presented as a timeline. Linux is chosen as a reasonable analog upon which to model research for OS X because it is the most similar platform to have received attention in the literature with regard to forensic memory analysis. However, note that Linux diverges significantly in terms of the kernel structures used to reconstruct information (Singh, 2006b).

An early paper on memory analysis for Linux proposes tools to aid embedded developers with identifying misused or wasted system memory (Movall, Nelson, & Wetzstein, 2005). While not well suited for forensic application, this work demonstrates how memory forensics follows as a natural extension of dynamic debugging. Burdach demonstrates this concept in a 2004 blog post on live forensic analysis for Linux systems. The blog notes acquisition of `/proc/kcore` should be favored over `/dev/mem` because its ELF core format allows for introspection using `gbd`. By comparing the addresses in `Symbol.map` with those from `/proc/kcore`, the author demonstrates how memory analysis can be used to perform rootkit detection. There is also a

description of how `/proc/kcore` can yield keyword searches with the UNIX strings command and regular expressions.

Moving beyond keyword searches and file carving techniques requires an understanding of operating system internals to determine how the kernel organizes information in what would otherwise be an incoherent block of data. Urrea (2006) began this work by enumerating many of the Linux structures relevant to forensic examination and shows how they might be extracted using a set of ridged Perl scripts. Burdach (2006) continued to enhance the analytic potential on the platform when he released a proof-of-concept toolkit called IDETECT to simplify introspection on memory, again using `gdb`. The same year FATKit framework (Petroni, Walters, Fraser, & Arbaugh, 2006) emerged as a major step forward by offering a modular approach to abstracting and visualizing forensic evidence from raw memory. Tool design is general and extensible enough that even in early releases FATKit featured modules for both Windows and Linux.

Case, Cristina, Marziale, Richard, and Roussev developed RAMPARSER to address "the lack of available memory parsing tools for Linux" (2008, p. S67). RAMPARSER can list running processes and perform introspection on specific ones to detail open files and socket information. Case, Marziale and Richard (2010) later extend RAMPARSER to make it "substantially less brittle with respect to kernel versions". This paper frames one of the major challenges to forensic memory analysis: most tools are specific to a particular platform version and must be reworked when even minor updates to an operating system occur. The constraint is especially dire for Linux distributions, where there is a great deal of kernel fragmentation. This same challenge is also the motivation for Foriana (Petroni, Walters, Fraser, & Arbaugh, 2006), a research tool

designed to explore solutions for memory analysis when the exact target OS version is not known. The tool has support for multiple architectures and operating systems, including BSD (from which Apple's UNIX foundation derives). Vidas (2011) suggests an open corpus for memory analysis to build support across a breadth of kernel versions.

While most of the work discussed exists primarily in the research domain, tools for Linux memory analysis are also represented commercially (Pikewerks Corporation, 2011). However, the open source forensics community has been slow to support Linux with tools built for users when compared to the capabilities available for Windows. In 2008 serious work began on this deficiency when the Digital Forensics Research Workshop (DFRWS) sought to develop tools specifically for Linux memory analysis. This emphasis resulted in several Linux extensions for the open source Volatility framework (Cohen & Collett, 2008). Case (2011a, b, c) writes extensively about subsequent work leading to the Linux branch available in beta from the Volatility project website. Integrated Linux support is anticipated for the 3.0 release of the project (M. Cohen, Volatility developer mailing list, March 26, 2012). In addition to the Linux commands shown in Table 1, Volatility has additional modules for analyzing network configuration, `kmem` cache, `dmesg` buffer, and auxiliary process details.

2.5.2 Volafox Project.

The first effort to provide higher-abstraction analytic capabilities for raw memory dumped on the Mac began with the open source release of volafox, a Google Code project described as a "Memory Analyzer for Mac OS X" (Lee, 2011). This text-based tool is implemented in Python 2.5 and borrows heavily from the Volatility source code. Volafox is operable on flat memory images captured from OS X 10.6-7 with 32 or 64-bit

Table 1. volafox modules and their Volatility equivalents.

volafox	OS X Terminal	Volatility (Linux branch)	Description
`os_version`	`sw_vers`		OS X build version
`machine_info`	`sysctl`	`linux_cpuinfo`	kernel version, CPU, and memory specifications
`mount_info`	`mount`	`linux_mount`	mounted filesystems
`kext_info`	`kextstat`	`linux_lsmod`	kernel extension (KEXT) list
`-m`			KEXT dump
`proc_info`	`ps`	`linux_task_list_ps`	process list
`-x`	`vmmap`	`linux_proc_maps`	process dump
`syscall_info`			syscall table with hooking detection
`net_info`	`netstat`	`linux_netstat`	network socket list
See Chapter 3	`lsof`	`linux_list_open_files`	open files list

architecture. Table 1 summarizes the volafox parsing modules and offers a comparison with those available for the Linux support branch of Volatility and tools built-in to the OS X command line. Little has been written about the usage of volafox, but Shuster (2011) describes its basic functionality in his blog.

Revision 52 of volafox, the version extended in Chapter 3, does not natively support the Mac Memory Reader (MMR) output format. Leat (2011) and ATC developer Hajime Inoue contributed to 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.

An alpha feature is also implemented in volafox to analyze network information, the output of which appears to be a simplified version of the UNIX `netstat` command. While difficult to ascertain the state of this module given the lack documentation, in

limited testing the feature only appears to support IPv4 TCP and UDP protocols. It is also unknown which kernel structures the module parses because this methodology is not included in the work by Suiche as with the others.

2.5.3 Symbol Table Construction.

In order to perform meaningful analysis of raw memory, an understanding of the composition and location of key kernel structures is required. Because Darwin (Apple's open source core for OS X) is freely available, 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" (2010). The KPCR structure can be used in Windows to get the symbols directly from memory, conveniently this structure is located at a static offset (Dolan-Gavitt, 2008b). Unfortunately, the same approach cannot be used on OS X because "kernel sections are destroyed as soon as the kernel (mach_kernel) is loaded by `removeKernelLinker()` function" [slides] (Suiche, 2010). A solution to the equivalent problem in Linux is addressed by Volatility, which maintains a database of overlay files containing the requisite symbol tables for select distributions and kernel versions (Case, 2011a). 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" (Suiche, 2010, p. 4).

Figure 1 shows key features of the `mach_kernel` executable file, located at the root directory of the OS X file system. Using the SYMTAB load command structure, a kernel symbol table can be constructed mapping the strings pointed to by

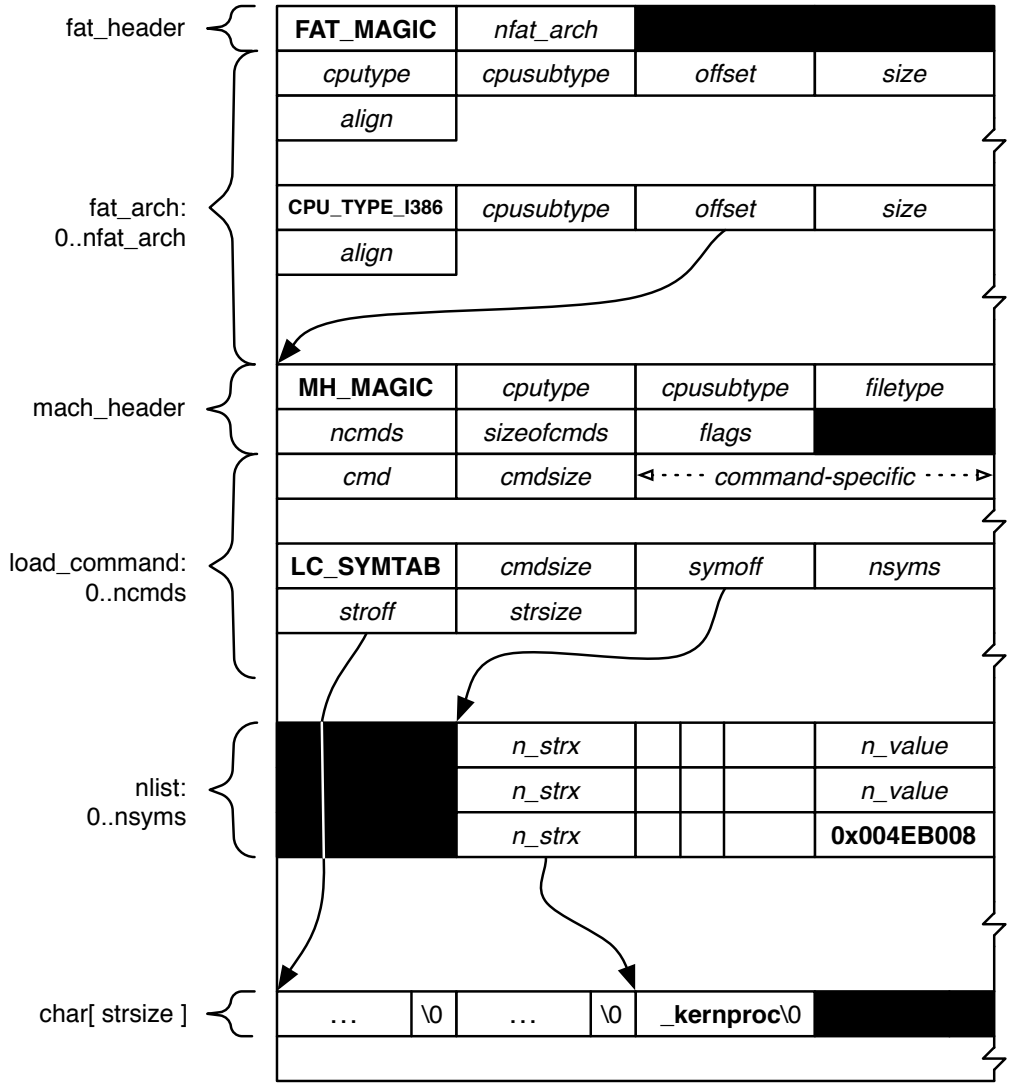


Figure 1. `mach_kernel` executable.

`symtab_command.stroff` with the static addresses stored in `nlist.n_value`. Such a table is valid for any installation sharing the same build version as the `mach_kernel` file used to derive it. This works because while the majority of physical memory is devoted to dynamic allocation for use by running processes, a portion of the layout is reserved for the kernel and its static data structures (Bovet & Cesati, 2006).

In revisions 25 and older, volafox built the symbol table directly from `mach_kernel` each the tool was executed. This was inefficient and also led to an undesirable dependency on the kernel executable. In practice, `mach_kernel` would therefore need to be copied from a target when performing memory acquisition. Leat provided a patch to the source, bringing the overlay functionality available in the Linux branch of Volatility to the volafox project (Leat, 2011; Lee, 2011). The current build (r52 at time of writing) includes an `overlay_generator.py` utility allowing symbol table generation for unsupported builds of OS X. A selection of overlay files is also being distributed with the project. However, since Apple does not publish a comprehensive list of OS X build numbers it is difficult to know whether this database is complete. A list of known builds is provided in Appendix A, but it is recommended that `mach_kernel` still be copied at time of memory acquisition to guarantee functionality.

2.5.4 Useful Structures.

The symbol table provides memory locations for a number of kernel structures useful in a forensic examination. The composition of these C structures (members and their types) can be determined via examination of the Darwin source code in which they are defined. Suiche (2010) describes several useful symbols and their associated structures as summarized in Table 2. This work mirrors core functionality of the volafox project so its equivalent module commands are also listed (Lee, 2011). Note the `proc` structure contains many additional members of interest to be explored in Chapter 3.

Table 2. volafox commands with associated symbols and kernel structures.

volafox Command	**os_version**
Kernel Symbol	version
struct Name	none (points to 100-byte string)
Source Definition	/xnu/osfmk/i386/lowmem_vectors.s
Useful Members	operating system, kernel version, date and username of compilation

volafox Command	**machine_info**
Kernel Symbol	machine_info
struct Name	machine_info
Source Definition	xnu/osfmk/mach/machine.h

Useful Members	integer_t major_version; // major kernel version
	integer_t minor_version; // minor kernel version
	uint64_t max_mem; // size of physical memory
	uint32_t physical_cpu; // number of CPU cores
	int32_t logical_cpu; // number of threads

volafox Command	**mount_info**
Kernel Symbol	mountlist
struct Name	mount
Source Definition	xnu/bsd/sys/mount_internal.h

Useful Members	TAILQ_ENTRY(mount) mnt_list; // linked-list of mounts
	struct vfsstatfs mnt_vfsstat;

struct Name	vfsstatsf
Source Definition	xnu/bsd/sys/mount.h

Useful Members	char f_fstypename; // file system type
	char f_mntonname; // directory mounted at
	char f_mntfromname; // mounted name

volafox Command	**kext_info**
Kernel Symbol	kmod
struct Name	kmod_info
Source Definition	xnu/osfmk/mach/kmod.h

Useful Members	struct kmod_info *next; // next linked module
	int id;
	char name[KMOD_MAX_NAME];
	char version[KMOD_MAX_NAME];
	int reference_count; // refs to this module
	vm_address_t address; // starting address
	vm_size_t size; // total size

volafox Command	**syscall_info**
Kernel Symbol	nsysent
struct Name	none (points to an int)
Source Definition	unknown
Useful Members	OS-dependent arithmetic required to find sysent based on the address of nsysent. Suiche (2010) provides details for OS X 10.5 and 10.6.
struct Name	sysent
Source Definition	xnu/bsd/sys/sysent.h
Useful Members	sy_call_t *sy_call; // implementing function

volafox Command	**proc_info**
Kernel Symbol	kernproc
struct Name	proc
Source Definition	xnu/bsd/sys/proc_internal.h

Useful Members	LIST_ENTRY(proc) p_list; // linked-list of procs
	pid_t p_pid; // process identifier
	pid_t ppid; // parent pid
	char p_comm[MAXCOMLEN+1]; // process name
	struct pgrp *p_pgrp

struct Name	pgrp
Source Definition	xnu/bsd/sys/proc_internal.h
Useful Members	struct session * pg_session;
struct Name	session
Source Definition	xnu/bsd/sys/proc_internal.h
Useful Members	char s_login[MAXLOGNAME]; // process username

The syscall table is potentially valuable for identifying hooked functions. Each system call addresses should match those stored on-file in `mach_kernel`, any discrepancies in the comparison could be evidence of tampering (Wowie, 2009). Location of the `sysent` structure in memory changes between versions of OS X, making the feature difficult to maintain. At present, the `syscall_info` command is broken in volafox for 10.7 captures due to this difficulty.

2.6 Summary

This chapter defined key terms and presented the processes for digital forensics, incident response, and memory analysis. Options available for capturing RAM on Intel Macs running OS X were discussed along with development and limitations of the Mac Memory Reader tool. A review of existing work in Linux memory analysis was then followed by a description of the volafox project being extended for this research. Technical details required in constructing a kernel symbol table on OS X and several useful kernel structures were also introduced.

III. Methodology

This chapter describes the 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 "get an understanding of files you should be concerned with during an investigation" (2009, p. 132). Because open files can help characterize process activity and highlight misuse of a computer, it is highly desirable to recover this information from memory.

A number of factors influence the decision to implement the feature selected. First, an open files listing is the first to-do item on the volafox project wiki. Second, this functionality is already represented in the Linux branch for Volatility (Table 1). Third, as described in Section 2.2.2, capturing this information is a recommended practice during incident response. Listing network connections was also considered due to the high forensic value of this information. However, because volafox already includes this as an alpha module, file handles were determined to have the greater research impact.

The chapter begins with the design goals of the system. Kernel structures responsible for the target information are then discussed. Organization of the volafox source is reviewed. Next, implementation of the new volafox module is described along with modifications to the exiting project source. Finally, the outstanding issues are listed.

3.1 System Design

The developed system consists of software for extracting the open files associated with processes from a raw memory image on OS X. Implementation extends the existing volafox tool to parse a variety of kernel structures not previously described in the literature. Object-oriented design provides a solution that is flexible with respect to both kernel architecture and operating system version.

3.1.2. Target Functionality.

The desired process-to-file handle information is an approximation of output from the commands `lsof` for UNIX and OS X (Apple Inc., 2011), or the Sysinternals equivalent `handle` (Russinovich, 2011) for Windows. Common to these tools is the process associated with each handle, a type classification, and unique identifier. The Linux branch of Volatility offers a possible format for this information as shown in Figure 2. Important to note in the output are the handle types for resources other than files on-disk, such as pipes used for inter-process communication (IPC) and network sockets.

```
PID: 2095   TASK: d1970550  CPU: 0   COMMAND: "gdm-binary"
ROOT: /    CWD: /var/gdm
 FD    FILE     DENTRY    INODE    TYPE  PATH
  0  d184b300  d14b9dd8  d19b90a0  CHR   /dev/null
  1  d14c4380  d14b9dd8  d19b90a0  CHR   /dev/null
  2  d0d0e500  d14b9dd8  d19b90a0  CHR   /dev/null
  3  d1520500  cf410338  d07ff9a8  SOCK  socket:/[6645]
  4  cca45f00  cbdfcb30  ccc025d8  PIPE
  5  d14c4ec0  cf422228  cf74bb28  PIPE
  6  d1407540  d150d800  d150ee40  CHR   /dev/console
  7  d1bd7380  cd59cf70  cf74b9d4  PIPE
  8  d14eb500  cf12add8  d0e83884  REG   /home/stevev/.xsession-errors
  9  d14ece40  cf12a888  cc0c821c  PIPE
 11  d150bd80  cbdfcb30  ccc025d8  PIPE
```

Figure 2. Volatility `linux_list_open_files` output (Cohen & Collett, 2008).

A second consideration in deciding on an output format is the resource available for validating the results. Because the `lsof` (list open files) command is included with OS X, it offers a convenient and reliable source of information for comparison. Emulating the output of this tool not only simplifies such analysis, it also gives the examiner a familiar interface to interact with. For these reasons, `lsof` was selected as the model for output format. Figure 3 shows sample output for the `lsof` command and Table 3 describes the information in each column. The new volafox module for listing open files includes functionality for parsing the nine default `lsof` fields present in Figure 3 and the mode identifier integrated with the FD column.

```
$ 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 3. UNIX list open files (`lsof`) command.

Table 3. 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.

3.1.3 Implementation Constraints.

While an ideal implementation would fully duplicate the functionality and nuances of the `lsof` command, the diversity of data structures required to accomplish this makes it impractical with the development resources available for this research. A design choice is made to focus on file rather than socket or IPC handles for this research due to the forensic value of the information and because file handles logically divide the development workload. Constraints of the tool implemented are formalized in Chapter 4, but there are two key decisions that influence the implementation.

First, the volafox open files module supports a subset of handle types, and only those subscribing to the virtual node (vnode) interface. The excluded types mean POSIX semaphores and shared memory files, kernel event queue files, pipes, and sockets 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 that the volafox open files module groups together using the generic description '`SOCKET`' in the TYPE field.

Second, the module supports a subset of the filesystems available for OS X, specifically 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` of 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 Key Kernel Structures

This section documents the kernel design recovery research objective specified in Section 1.1. Implementing the new volafox module for listing open files requires knowledge of 32 unique C data structures from the OS X source code, four of which are described by Suiche (2010) 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 kernel isn't heavily commented and its internals aren't documented, so you learn by tracing code by hand" (Sesek, 2012). Figure 4 shows an overview of the relevant structures and associated UNIX `lsof` fields from Table 3. The names, source files, interesting members, and relationships between these data structures appear as series of relationship diagrams in Figures 5-8 and Appendix B.

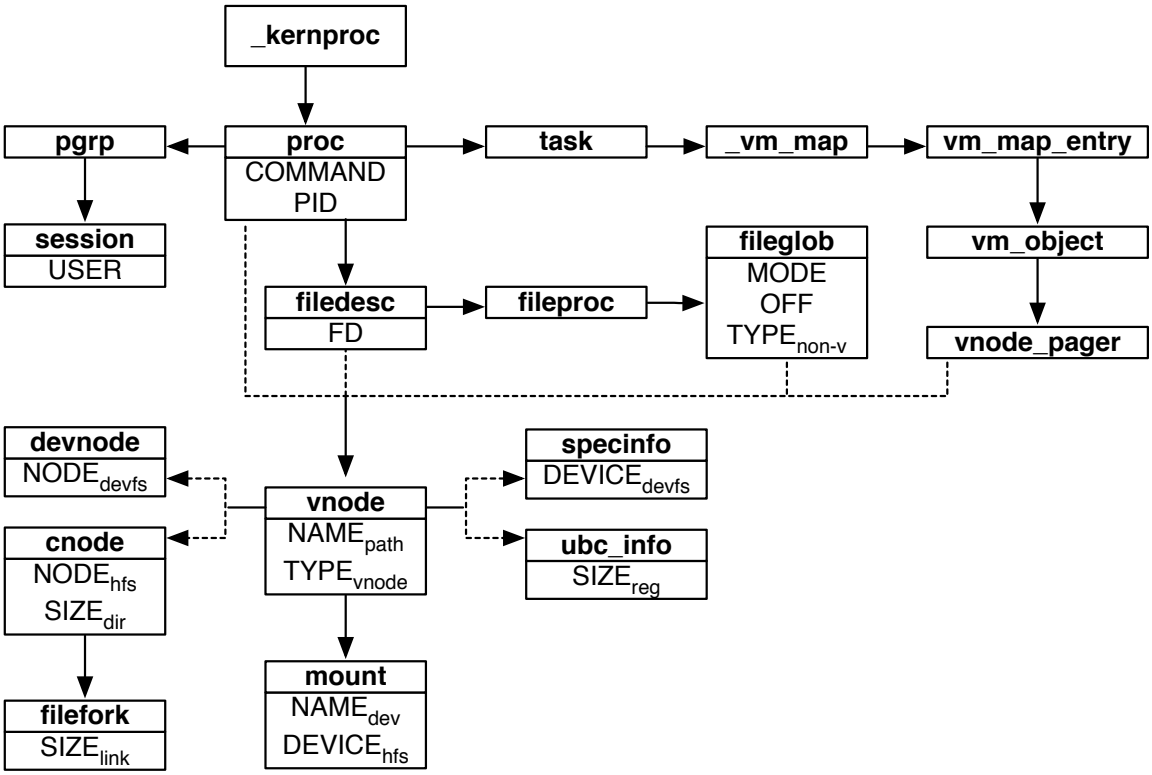


Figure 4. C struct relationship overview.

Extracting open file information begins with the kernel symbol table, shown in Figure 5 and described in Section 2.5.3. Symbol `_kernproc` provides an address for the head of the process list, `kernel_task` (PID 0), which is unique in its use of static data structures (Singh, 2006b, p. 293). Because of this property, PID 0 does not appear in the output of UNIX commands such as `ps` or `lsof` and therefore is excluded from the file handles module implementation. The COMMAND and PID fields are members of `struct proc`, and USER is located in `struct session`. Note that `p_list` is a substructure, meaning `proc` contains it as a member rather than using a pointer to reference it.

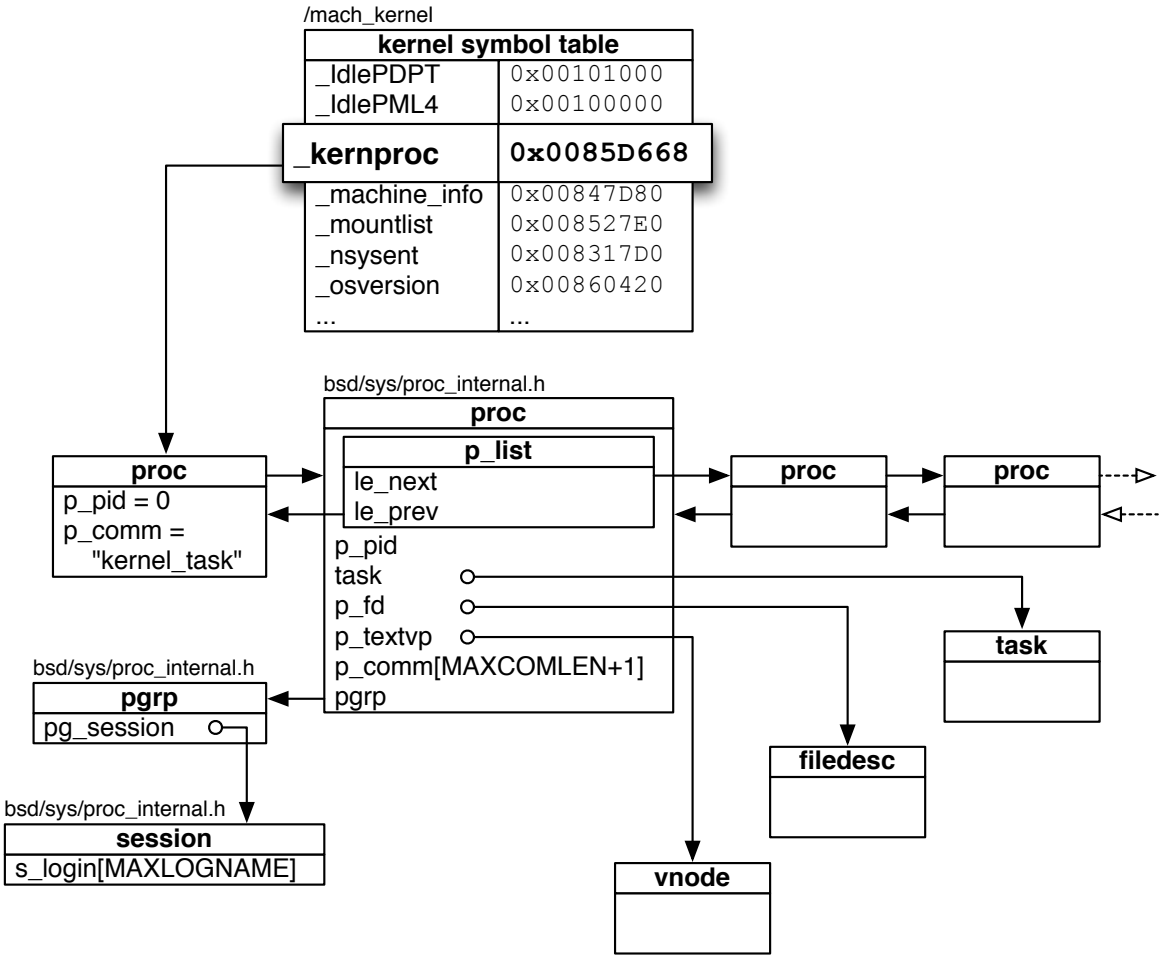


Figure 5. Symbol table and process list.

Structure `task`, as pointed to by `proc` in Figure 5, provides the link to program text files (FD `txt`). Program text refers to the code and data segments of an executable, as well as linked libraries and any other memory-mapped files. This information is invaluable for identifying rouge processes, which often masquerade as legitimate executables to avoid detection (SANS Institute, 2008).

Figure 6 shows how these files are referenced. Structure `_vm_map` stores the header for a ring of `vm_map_entry` structures containing the union member `object`. 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, 2006b, p. 571).

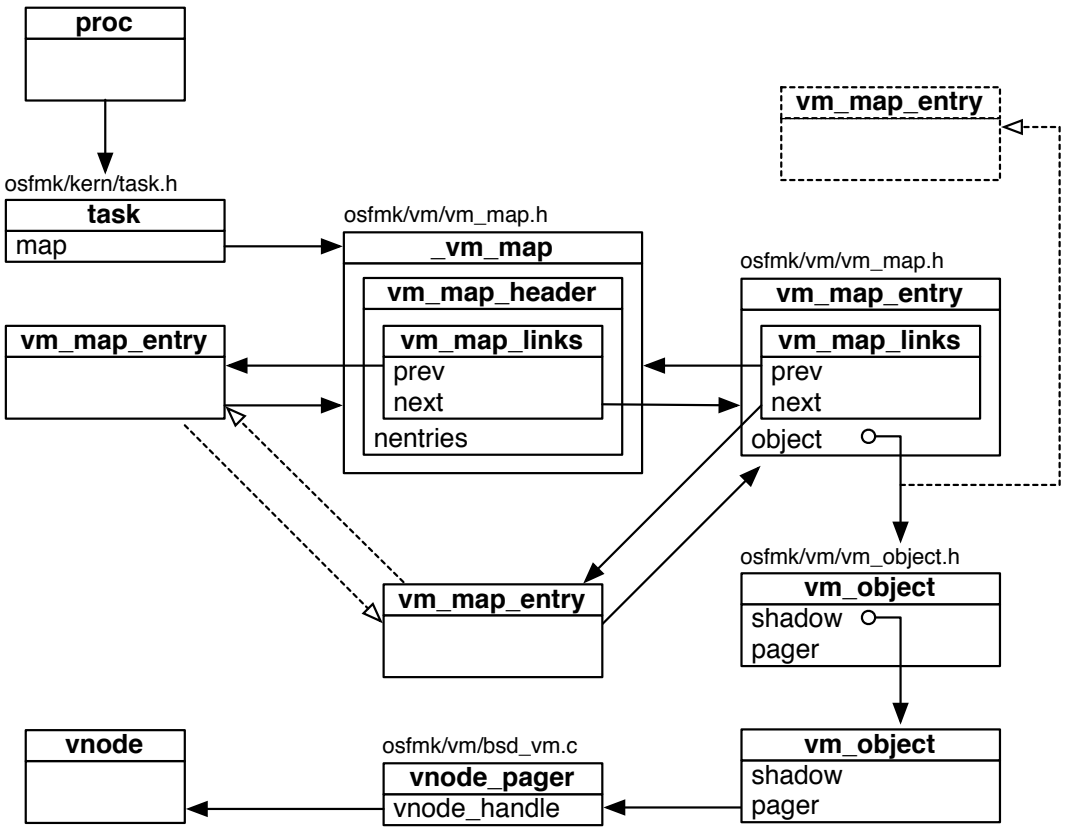


Figure 6. Memory-mapped files (`txt`).

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 index into this array represents the numerical file identifier used by the FD field of the `lsof` output. 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.). File mode, or read/write access, is determined from the value of `fileglob.fg_flag` using the bitmap definitions in `bsd/sys/fcntl.h`. Member `fileglob.fg_offset` is the offset for FIFO and special character files as reported in the `lsof` SIZE/OFF field. The

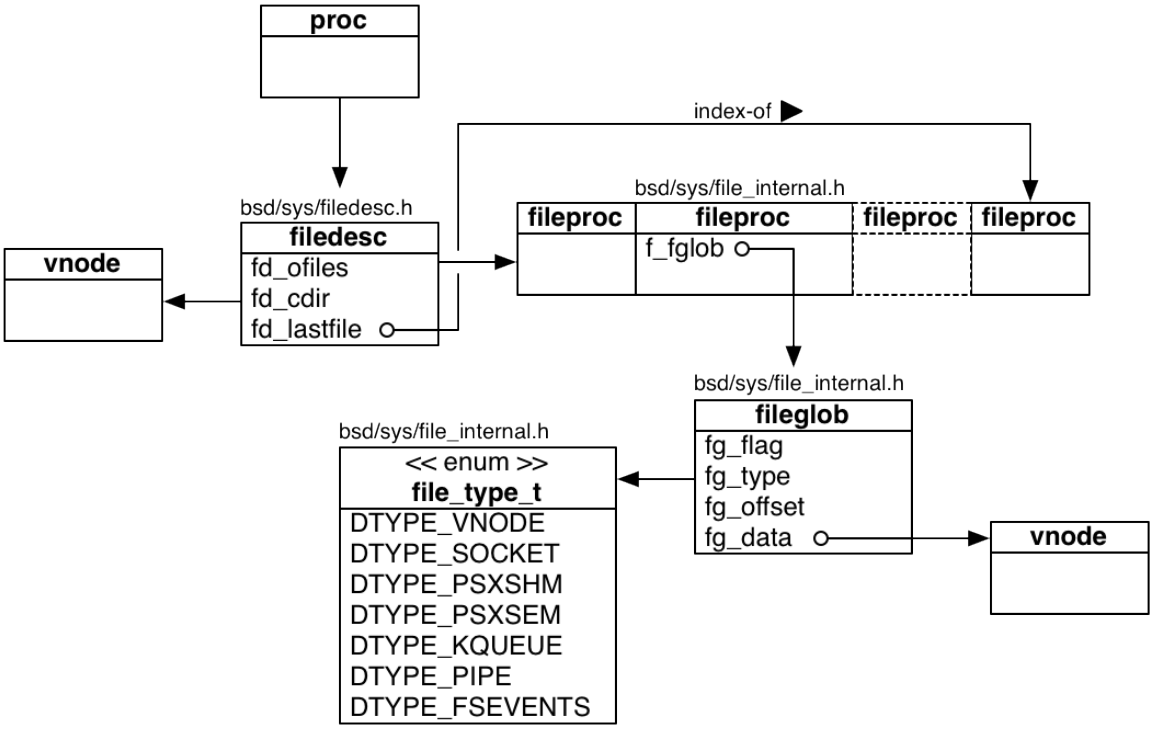


Figure 7. File descriptor table.

`fileglob.fg_data` member is a generic pointer that may reference a variety of structures. As described in Section 3.1.3, full support is constrained to vnode types. Enumerator `fileglob.fg_type` holds destination type of the pointer. Any non-vnode handle can be typed using the values of `enum file_type_t`, but DEVICE, SIZE/OFF, NODE and NAME fields are unsupported for such handles using this research implementation.

Figure 8 shows the `struct vnode` referenced by `proc`, `vnode_pager`, `filedesc` and `fileglob` structures shown in Figures 5-7. The `lsof` NAME field is a concatenation of the device name from `mount.vfsstatfs.f_mntfromname` and a path of `vnode.v_name` strings recursively references using `vnode.v_parent`. Member `vnode.vtype` describes the file type of any supported file, and `vnode.v_tag` holds the associated filesystem. The number of combinations created by `vnode.v_type` and `vnode.v_tag` leads to branches at `union v_un` and the generic pointer `v_data`. NODE is stored at `devnode.dn_ino` for all files using the VT_DEVFS filesystem and in `cnode.cat_desc.cd_cnid` for VT_HFS. An encoded device identifier is found at `specinfo.si_rdev` for VT_DEVFS and in `mount.vfsstatfs.fsid.val[0]` for VT_HFS. The device identifier is decoded using macros in `bsd/sys/types.h` that return major and minor device numbers. Returning the correct `lsof` SIZE/OFF value requires knowledge of `vnode.v_type`. For VREG files the size is found in `ubc_info.ui_size`, however this structure is not valid for system vnodes that are otherwise regular (Singh, 2006b, p. 605).

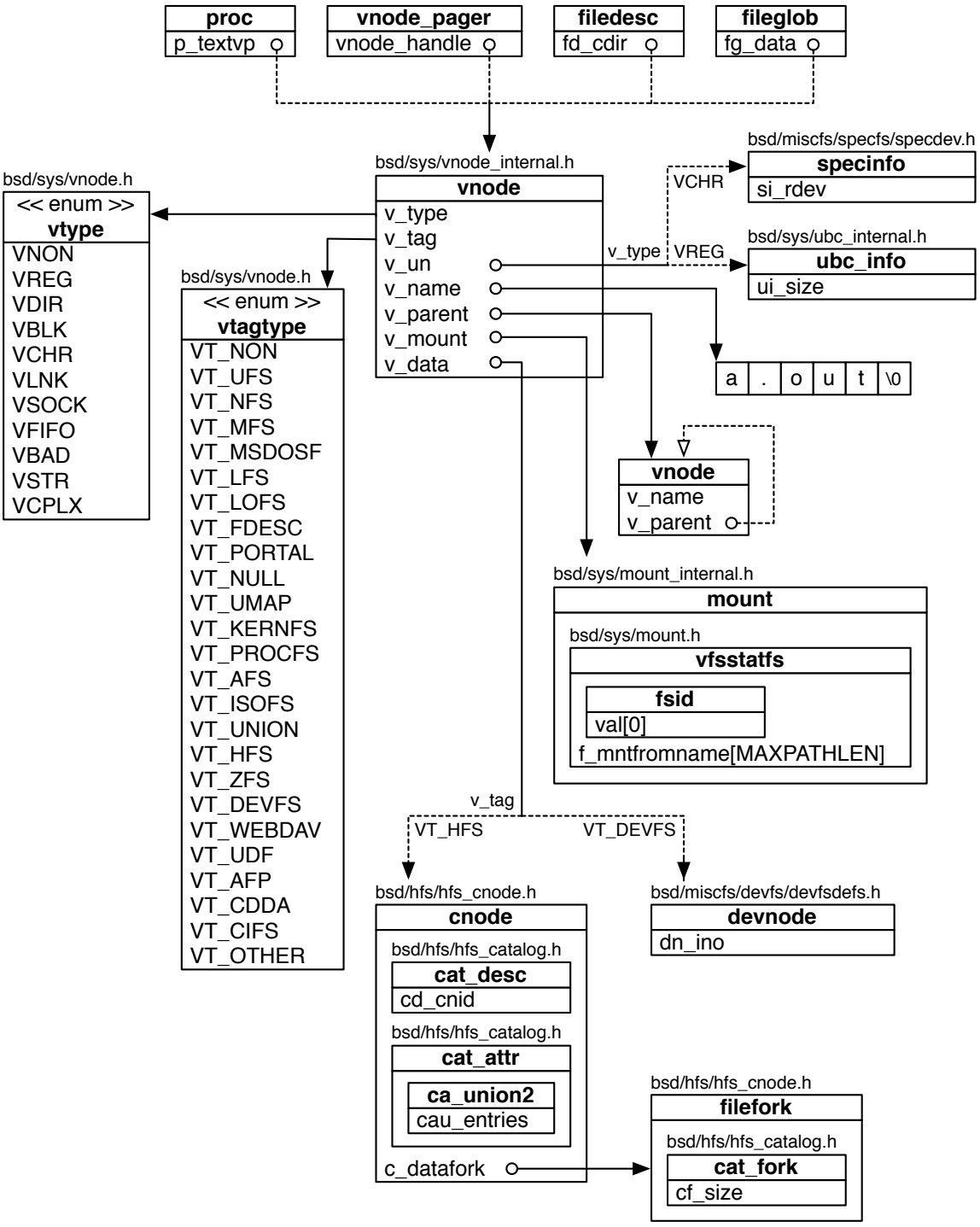


Figure 8. Virtual node (`vnode`).

VDIR file size is calculated using the count in `cnode.cat_attr.cau_entries`, the equation:

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

found in `bsd/hfs/hfs_vnops.c`, and the macro definition from `bsd/hfs/hfs.h`. Finally, VLNK sizes are located in `filefork.cat_fork.cf_size`.

Starting with the address pointed to by `_kernproc`, this section describes the structures and relationships for retrieving information needed to emulate `lsof` output for all vnode type files stored on HFS+ or DEVFS filesystems. Table 4 summarizes the structure members needed to support the required fields.

Table 4. Open file data locations.

	! DTYPE _VNODE	DTYPE_VNODE				
		VT_HFS				VT_DEVFS
		VREG	VDIR	VLNK	VFIFO	
COMMAND	proc.p_comm					
PID	proc.p_pid					
USER	session.s_login					
FD & mode	filedesc.fd_ofiles[i] + fileglob.fg_flag					
TYPE	fileglob. fg_type	vnode.v_type				
DEVICE		mount.vfsstatfs.fsid.val[0]				specinfo. si_rdev
SIZE/ OFF		ubc_info. ui_size	cnode. cat_attr. cau_entries	filefork. cat_fork. cf_size	fileglob. fg_offset	
NODE		cnode.cat_desc.cd_cnid				devnode. dn_ino
NAME		mount.vfsstatfs.f_mntfromname + recurse(vnode.v_parent->v_name) + vnode.v_name				

3.2.1 OS X Abstraction Layers.

OS X is a commercial operating system integrating a collection of technologies, a subset of which are made open source by the Darwin UNIX distribution. This research is concerned with data structures defined by the Darwin kernel, known as xnu. The xnu kernel is composed of several additional abstraction layers. Mach, sometimes described as the xnu microkernel, provides critical low-level services. These are leveraged by BSD, "the primary system programming interface" (Singh, 2006b, p. 31). Among other features, the BSD layer supports a process model and virtual file system (VFS) layer. BSD hooks into Mach for numerous services, such as task operations responsible for execution and the virtual memory subsystems (Singh, 2006b, p. 33).

Much of the preceding design recovery is a consequence of manual inspection of the Darwin source code and headers, combined with prototype development in volafox to verify the purpose of various structure members. However, there are several instances where the destination of a pointer reference is of unknown or ambiguous type, complicating this analysis considerably. Figure 9 demonstrates the problem using `struct proc`. While three of the members shown are explicitly typed, such as `pid_t p_pptr`, the structure pointed to by `task` is unknown using the definition alone. Similar issues occur at `vm_object.pager`, `fileglob.fg_data`, `vnode.v_un`

```
struct proc {
      LIST_ENTRY(proc) p_list;
      pid_t            p_pid;
      void *           task;  /* corresponding task (static)*/
      struct proc *    p_pptr;
      …
};
```

Figure 9. `struct proc`.

and `vnode.v_data`. These generic object pointers appear at locations where an interface is needed between two or more different abstraction layers. In the example, `struct proc` is a BSD structure and `proc.task` points to a Mach structure. Figure 10 summarizes the relevant layer interfaces. Note that while `fileglob.fg_data` only branches back into the BSD layer for this implementation, support for additional file types would involve structures such as those for pipes and sockets that may exist outside the BSD abstraction layer (Singh, 2006b, p. 919).

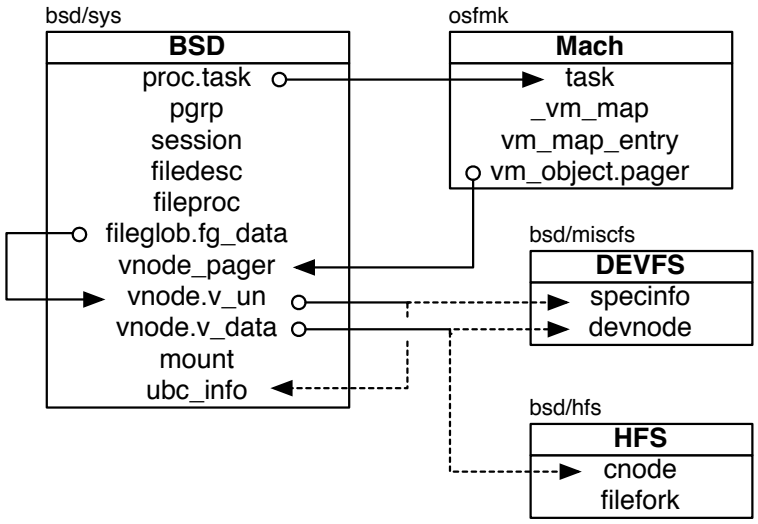


Figure 10. Abstraction crossover.

3.3 Project Volafox

This section describes software design details of the volafox project needed to understand implementation of the new open file listing module. First, the relevant source files and Python classes are introduced. Next, the execution flow for the main project file is traversed to explain where the new module interfaces with the existing code.

3.3.1 Package Organization

Figure 11 shows a summary of the source files from the volafox package related to OS X memory analysis. Public classes in each file are indicated 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 3.4.

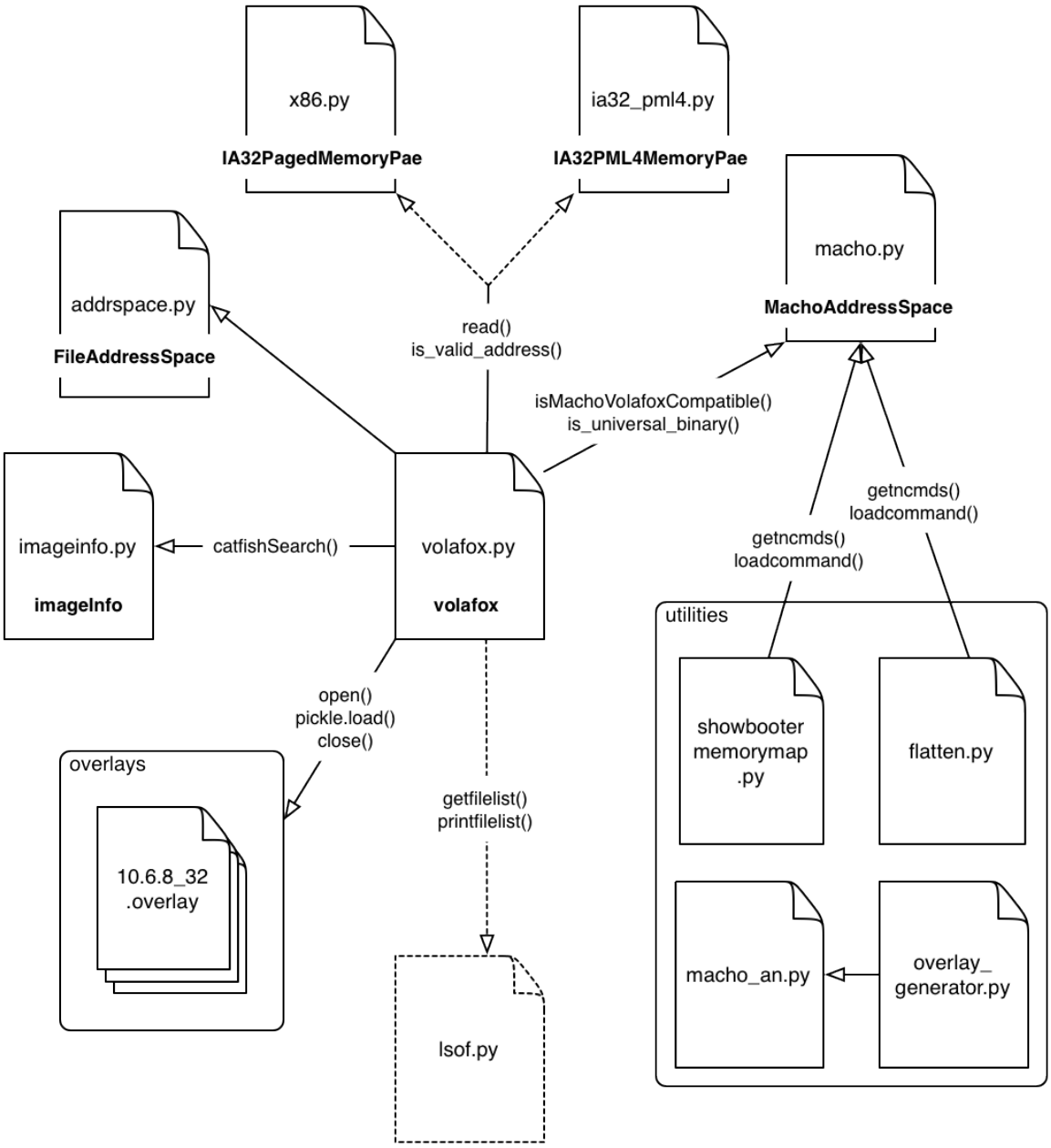


Figure 11. volafox package diagram.

Main program execution depends on the following source files and directory:

`addrspace.py` – houses `FileAddressSpace` class responsible for file operations on linear memory images.

`macho.py` – support for the Mac Memory Reader image format (Mach-O), the `MachAddressSpace` class is written to be the MMR format equivalent of `FileAddressSpace`. As of revision 48 of the project this functionality was disabled due to compatibility problems with 64-bit analysis.

`x86.py` / `ia32_pml4.py` – these 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 either `FileAddressSpace` or `MachAddressSpace` (whichever is passed to the initializer). 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 (Intel Corporation, 2012), meaning the second file is the one responsible for handling 64-bit architecture images where the first is used for 32-bit.

`imageinfo.py` – the `imageInfo` class inspects a memory image file to determine the file format (MMR or linear) and kernel architecture (32 or 64-bit) required to initialize the correct address space and PAE objects already described. It also returns the OS X build version information for the image needed to select the correct overlay file. This file also has a main so it can be executed as a stand-alone utility.

Usage: `$ python imageinfo.py IMAGE.mem`

`overlays/` – as of revision 48, `volafox.py` no longer accepts a `mach_kernel` file argument for building the symbol table. All symbols are read from files in the overlays directory labeled by OS version and architecture using the Python `pickle` library for object serialization. New overlays can be generated from the kernel executable with the `overlay_generator.py` utility.

`volafox.py` – houses the project `main()` and `class volafox` responsible for marshaling the remaining files and classes to preform analysis of OS X memory images.

Usage:
```
$ python volafox.py -i MEMORY_IMAGE
        [-o INFORMATION][-m KEXT ID][-x PID]
```

The volafox package also includes several stand-alone utilities:

`showbootermemorymap.py` – outputs the load commands from an MMR image in the same format as the `showbootermemorymap` kernel macro debug script and the `/dev/pmap` device.

Usage: `$ python showbootermemorymap.py IMAGE.mmr`

`flatten.py` – converts MMR image files from Mach-O into a linear equivalent which can be analyzed by `volafox.py`. This script is only operable on 32-bit architecture, as verified using the `imageinfo.py` utility.

Usage: `$ python flatten.py SOURCE.mmr DEST.flat`

`overlay_generator.py` – reads the symbol table from a `mach_kernel` executable and stores to file in the form of a serialized Python dictionary for use in the `overlays` directory.

Usage:
```
$ python overlay_generator.py MACH_KERNEL
10.MAJOR.MINOR_ARCH.overlay [32|64]
```

3.3.2 Module Interface.

As summarized in Table 1, volafox features a number of command options for parsing information from raw memory. The code implementing these branches, around 1300 lines, is contained within the source file `volafox.py`. This monolithic software design is not particularly modular, but the implementation of the new open files functionality strives to be. This section explains where the new code interfaces with the existing project.

Since `volafox.py` is intended to be run in Python executable mode, the support code of concern begins in `main()`. This function performs the following actions:

1. Handle command line arguments and the usage statement.
2. Instantiate a new `volafox` object with path to the raw memory image file.

3. Delegate to `volafox` object for initialization of the correct address space and PAE objects, a method that also returns the architecture and OS version needed to select the correct overlay file.

4. Import a symbol table as a Python dictionary from the correct overlay stored on file as a serialized object.

5. Pass addresses for the `_IdlePDPT` and `_IdlePLM4` symbols to the `volafox` object, which uses them to initialize the page table map and thereby completes setup for virtual to physical address translation.

6. Pass address for the `_machine_info` symbol to the `volafox` object, which uses it to determine the kernel version and stores the result as an instance variable for branching based on OS (Lion versus Snow Leopard).

7. Branch based on user information requested to call the appropriate `class volafox` method. Each information method accepts a kernel symbol address and returns a string matrix of results.

8. Print results and exit.

The new module adds code to `main()` for additional argument handling and a branch for calling a new `lsof` method in `class volafox`. Method `lsof` branches on architecture to correctly read and unpack the `_kernproc` symbol and delegates all other open file analysis to the new source file `lsof.py` shown in Figure 11.

3.4 File Handle Module Implementation

This section introduces a new file handle module for volafox revision 52. First, the Unified Modeling Language (UML) is used to present a graphical view of how the new `lsof.py` source file is organized in Figures 12-13 and Appendix E. The solution to flexible analysis of multiple kernel architecture and OS versions follows. Next, the issue of ambiguous data types is discussed. Finally, modifications to the existing volafox source code are listed.

3.4.1 Object-oriented Design.

While the open files module does not require a complex inheritance hierarchy, a class diagram still offers the best visual explanation of the software's design. UML is therefore used to provide an overview of the `lsof.py` source file, the complete version of which is available as a single graphic in Appendix D. Some liberty has been taken with the standard since the source integrates both object-oriented and imperative programming elements. Note that aggregate associations are modeled when a class definition includes an explicit instance variable of another class, while dependencies are used if a class constructs instances for use only within method scope.

Figure 12 shows the class elements corresponding to structures of the process list and the file descriptor table. It also specifies the abstract superclass `Struct`, the parent of all remaining classes in `lsof.py`. Figure 13 covers structures related to the vnode interface and memory-mapped files. The utilities box describes global variables and function dependencies outside the class hierarchy. The `lsof.py` box shows imperative functions which depend on the classes, including the public `getfilelist()` and `printfilelist()`, which serve as an interface to the remaining volafox source code.

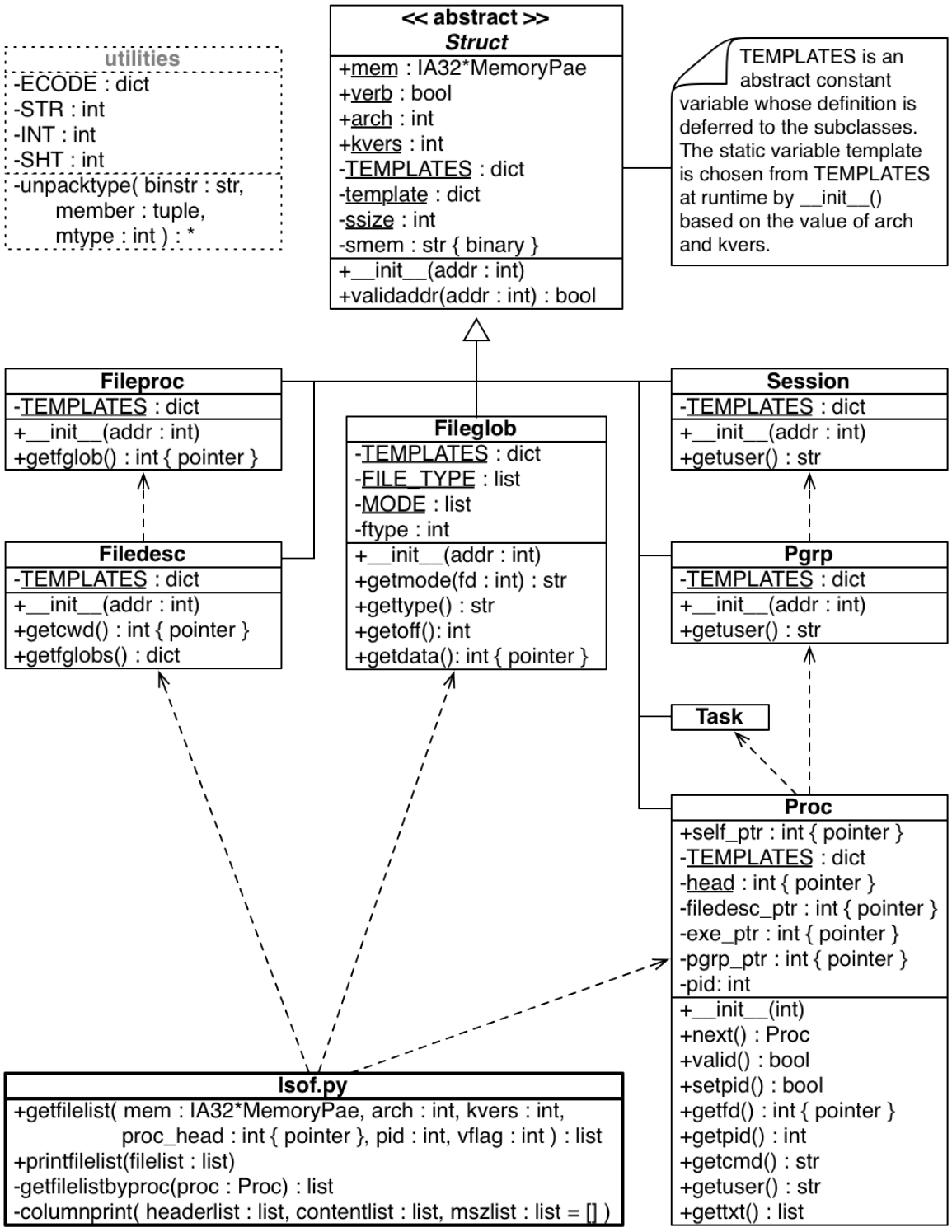


Figure 12. UML 1 – process list and file descriptors.

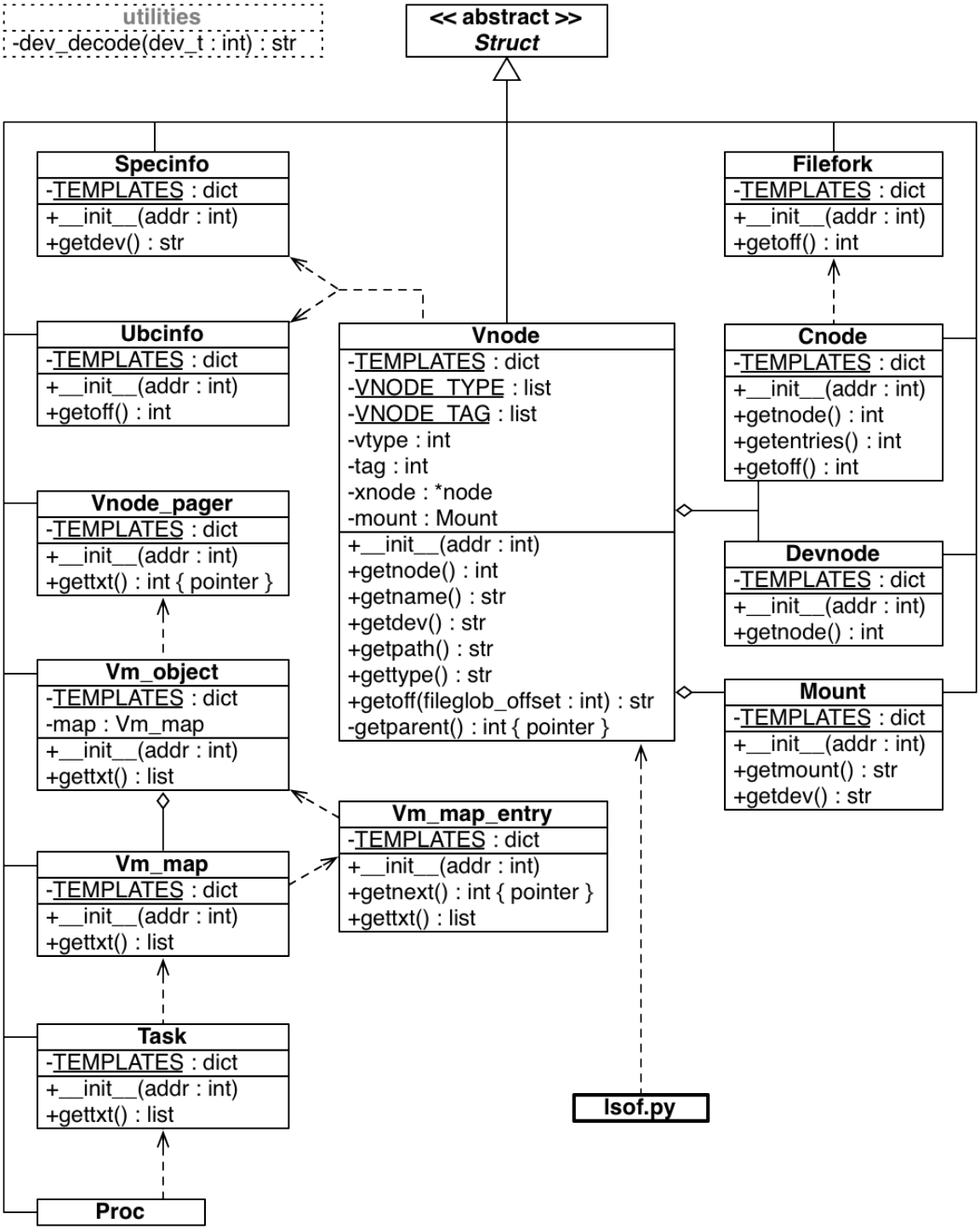


Figure 13. UML 2 – vnode interface and memory-mapped files.

Though Python is not a *strictly* object-oriented language, its support for classes allows for a modular implementation that maps to the network of kernel structures previously discussed. Because C structs group data types, a natural solution to parsing their content uses object instances as containers for the address space they occupy and class methods for handling the unpacked data (see Appendix C. Python `struct` Library). The open files module defines classes for 18 of the structures described in Section 3.2, the remaining substructures are handled inside the class methods since they occupy the address space of the struct in which they are defined.

3.4.2 Structure Templates.

Sections 3.4.2 - 3.4.5 collectively document the research objective from Section 1.1 that requires a flexible process for programmatically handling kernel data structures defined for different kernel architectures and operating system versions. 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. The module implemented for this research uses a dynamic runtime solution consisting of two parts.

First, the following interface is defined to describe required members of each structure for a given architecture and OS version:

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

Table 5. 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*)

Table 5 lists Python types from the C struct 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 14 shows the 32-bit Snow Leopard variant for the `struct proc` template.

```
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 14. `struct proc` template, 10.6 x86.

The second component in the template solution is a Python class initializer that dynamically selects the correct template for a given subclass at runtime 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 15.

```
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 15. Simplified abstract `class Struct` (no error handling).

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 described in Section 3.3.1. 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.

Combining the structure template interface with an abstract initializer offers a solution that greatly simplifies the 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 16 shows the concrete subclass corresponding to `struct devnode` and demonstrates use of the structure template solution.

```
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 16. Concrete class Devnode.

3.4.3 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 open files module uses *(18 classes * 2 versions * 2 architectures) = 72* struct 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 17 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 5 and `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.

```
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 17. Template generation function.

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 the structures described in Section 3.2 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 the following steps applied to each required header:

1. Identify all required definitions in a given header file, cut the file content after the last statement needed to avoid irrelevant dependencies.
2. Remove any `#ifdef` macros necessary to expose the target definitions.
3. Remove any `#include` statements for kernel dependencies and replace with a local version of each header file containing a required definition.
4. Recursively apply steps 1-4 for each local header added.
5. Troubleshoot ad nauseam until the target header can be included without compilation error.

These steps must be completed for each supported version of OS X due to subtle changes within the header files. For the 10.7 version of the offsets program 27 different header files were required to define the data structures needed by the open files module. The

10.6 version uses only 17 headers because many of the required definitions are relocated to the dependent file rather than including them recursively.

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. Figure 18 shows the `offsets.c` template contradicting the value calculated manually from the structure definition in Figure 19.

```
struct_ubcinfo = {'ui_size':('off_t',32,8,'SIZE/OFF')}
```

Figure 18. 10.7 x64 template for `struct ubc_info`.

```
struct ubc_info {
      memory_object_t         ui_pager;   // 8-byte pointer
      memory_object_control_t ui_control; // 8-byte pointer
      uint32_t                ui_flags;   // 4-byte int
                                          // 4-byte PAD (ptr alignment)
      vnode_t                 ui_vnode;   // 8 byte pointer
      kauth_cred_t            ui_ucred;   // 8 byte pointer
                                          // --------------
                                          // 40-byte offset

      off_t                   ui_size;    // 8-byte int
      ...
      };
```

Figure 19. Manual offset calculation for 64-bit `struct ubc_info` (annotated).

Table 6. Manual hex sizing.

ubc_info	ui_pager		ui_control	
hex	A03F4207	80FFFFFF	C02B3B07	80FFFFFF
human	ffffff8007423fa0		ffffff80073b2bc0	
ubc_info	ui_flags		ui_vnode	
hex	1F000000	00000000	00104007	80FFFFFF
human	31		ffffff8007401000	
ubc_info	ui_ucred		ui_size	
hex	00000000	00000000	00EE1400	00000000
human	NULL		1371648	

The `printhex()` utility written for the open files module can be used to print the address space in hex of a problematic structure for debugging. Output can be used to confirm manual sizing as shown in Table 6. The correct value of `ui_size` in the example can be verified using output from the UNIX `lsof` command on the machine the memory was captured from to be sure the interpretation is correct. This analysis indicates `ui_size` should have a 40-byte offset instead of 36 in the template for `struct ubc_info`. Similar offset issues exist in the `offset.c` templates for `struct vnode_pager` and `struct task`. Manual offset calculation and hex analysis was effective in resolving the problem for these templates as well. In all three cases, the solution is a manual 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 and also print the structure members in a human-readable format for ease of debugging. Figure 20 shows the output of the program. The architecture argument on the command line instructs `printstructs.py` to compile `offsets.c` using `gcc -arch` with either `i386` or `x86_64` specified as

```
$ ./printstructs6.py 64 -p
struct_proc = {'p_list':('LIST_ENTRY(proc)',0,16,'',{'le_next':( [...]

      ----------- struct proc (6x64) -----------
      LIST_ENTRY(proc) p_list   0 16
        struct proc *le_next    0  8
        struct proc **le_prev   8  8
      pid_t p_pid              16  4 PID
      void *task               24  8
      struct filedesc *p_fd   200  8
      struct vnode *p_textvp  664  8
      char[] p_comm           700 17 COMMAND
      struct pgrp *p_pgrp     752  8
```

Figure 20. Template output from `printstructs.py`.

appropriate. The `-arch` flag is an Apple-only option according to the `gcc` manpage, though in limited testing the standard `-m32` and `-m64` flags also appear to work. Setting the flag allows one version of `offset.c` to produce templates for both the 32 and 64-bit installations. Dictionary output from `printstructs.py` was then pasted into the `TEMPLATES` constant of each class in `lsof.py` to complete the definition.

3.4.4 Unions and Type Ambiguity.

As discussed in Section 3.2.1, the union data structure and generic pointers can lead to ambiguity which must be well-handled by the open files module. One example is the `vm_object` member of `struct vm_map_entry`. This union may refer to a pointer for `vm_object` or `_vm_map` and the address itself cannot be used to distinguish the two. The open files module deals with this situation by implementing a *template test*, which unpacks the values at certain offsets and attempts to match these with the types expected for the template of a particular structure. The initializer for `class Vm_object` demonstrates the technique in Figure 21.

The vm_object object structure is characterized by two pointer members followed by a lock, which is in contrast to the lock followed by two pointers in the _vm_map definition. Moreover, vm_object appears to initialize these pointers with the address of the object itself, suggesting both should always be valid when tested. The template test provides a reliable indicator of the structure type for the pointer stored at vm_object.vm_map_entry and is used to branch program logic.

```
class Vm_object(Struct):

  TEMPLATES = {...}

  def __init__(self, addr):
    super(Vm_object, self).__init__(addr)
    self.map = None

    ptr1 = unpacktype(self.smem, self.template['memq'][4]['next'], INT)
    ptr2 = unpacktype(self.smem, self.template['memq'][4]['prev'], INT)

    if ptr1 == 0 or ptr2 == 0 \
      or not(Struct.mem.is_valid_address(ptr1)) \
      or not(Struct.mem.is_valid_address(ptr2)):

      # on failure, create map instance to be called recursively
      self.map = Vm_map(addr)
```

Figure 21. Template testing.

3.4.5 Modifications to `volafox.py`.

The implementation for listing open files is confined to the source file lsof.py whenever possible in order to support modular software design. However, three noteworthy changes are required to integrate new code with the existing volafox.py source file. First, additional argument handling is needed to direct execution of the open files module from the command line. The modified volafox usage statement in Figure 22 explains the new options.

```
$ python volafox.py

volafox: release r52; lsof research build 1.0
project: http://code.google.com/p/volafox
   lsof: osxmem@gmail.com
support: 10.6-7; 32/64-bit kernel
  input: *.vmem, *.mmr (Mac Memory Reader, flattened x86)
  usage: python volafox.py -i IMAGE [-o COMMAND [-vp PID]]
                                    [-m KEXT_ID][-x PID]

WARNING: this is an experimental development build adding support for
listing open files. The code here is NOT in sync with project trunk.

Options:
-o CMD : Print kernel information for CMD (below)
-p PID : List open files for PID (where CMD is "lsof")
-v     : Include unsupported types in listing (where CMD is "lsof")
-m KID : Dump kernel extension address space for KID
-x PID : Dump process address space for PID

COMMANDS:
os_version     Mac OS X build version
machine_info   kernel version, CPU, and memory specifications
mount_info     mounted filesystems
kern_kext_info kernel KEXT (Kernel Extensions) listing
kext_info      KEXT (Kernel Extensions) listing
proc_info      process list
syscall_info   syscall table
net_info       network socket listing (hash table)
lsof           open files listing by process (research)
```

Figure 22. volafox usage statement.

The optional `-p` flag emulates the UNIX `lsof` command option to print open files only for a specified process. Because the new module only fully supports the handle types described in Section 3.1.3, the `-v` flag was added to avoid output of limited value to the examiner by default. The code supporting the new options is of generic, imperative design is therefore not discussed in detail.

Figure 23 shows the second modification, a new module branch added to `main()`, note the call to `printfilelist()`, one of two public functions in `lsof.py`.

```
elif oflag == 'lsof':
  filelist = m_volafox.lsof(symbol_list['_kernproc'], pid, vflag)
  if vflag:
    print ""
    printfilelist(filelist)
    sys.exit()
```

Figure 23. volafox lsof command branch.

Finally, Figure 24 shows the stub method added to `class volafox` which calls the second public function `getfilelist()`.

```
def lsof(self, sym_addr, pid, vflag):

  if self.arch == 32:
    overlay starting at symbol _kernproc
    kernproc = self.x86_mem_pae.read(sym_addr, 4);

    proc_head = struct.unpack('I', kernproc)[0]

  else: # 64-bit
    kernproc = self.x86_mem_pae.read(sym_addr, 8);
    proc_head = struct.unpack('Q', kernproc)[0]

  return getfilelist(self.x86_mem_pae, self.arch, self.os_version, \
                     proc_head, pid, vflag)
```

Figure 24. lsof method stub added to class volafox.

3.5 Open Issues

While the majority of system design goals outlined for the open files module in Section 3.1 are met by the implementation described in 3.4, there are two outstanding deficiencies not accounted for in the constraints specified. Both describe limitations of the kernel structure analysis rather than programming error. Problems reporting the correct process user and sizing the `/dev` directory are discussed in this section.

3.5.1 User Field Reporting.

The manpage for the UNIX `lsof` command describes the 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 25-26.

```
$ ./volafox.py -i 10.6.8x86.vmem -o lsof -p 15
COMMAND   PID   USER   FD      TYPE     DEVICE  SIZE/OFF   NODE […]
distnoted  15   root  cwd       DIR       14,2      1088      2 […]
distnoted  15   root  txt       REG       14,2     50672 268317 […]
distnoted  15   root  txt       REG       14,2   1054960 264388 […]
distnoted  15   root  txt       REG       14,2 213385216 466405 […]
distnoted  15   root   0r      CHR        3,2       0t0    297 […]
distnoted  15   root    1      PIPE         -1        -1     -1 […]
distnoted  15   root    2      PIPE         -1        -1     -1 […]
distnoted  15   root   3u    KQUEUE         -1        -1     -1 […]
distnoted  15   root   56u   SOCKET         -1        -1     -1 […]
```

Figure 25. volafox user output.

```
# lsof -p 15
COMMAND   PID     USER   FD     TYPE     DEVICE  SIZE/OFF   NODE […]
distnoted  15   daemon  cwd      DIR       14,2      1088      2 […]
distnoted  15   daemon  txt      REG       14,2     50672 268317 […]
distnoted  15   daemon  txt      REG       14,2   1054960 264388 […]
distnoted  15   daemon  txt      REG       14,2 213385216 466405 […]
distnoted  15   daemon   0r      CHR        3,2       0t0    297 […]
distnoted  15   daemon    1     PIPE 0x02fe2af8     16384        […]
distnoted  15   daemon    2     PIPE 0x02fe2af8     16384        […]
distnoted  15   daemon   3u  KQUEUE                             […]
distnoted  15   daemon   56u   unix 0x02fdef80       0t0        […]
```

Figure 26. `lsof` user output.

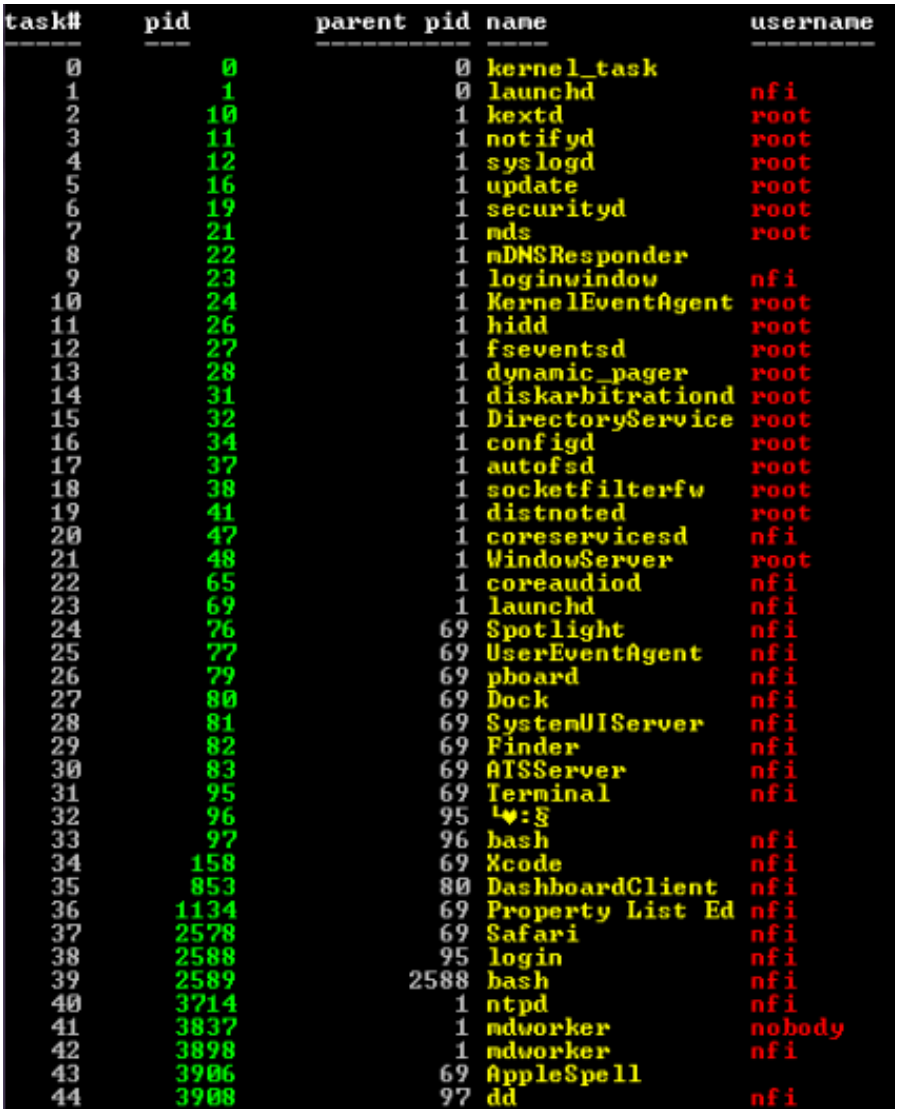
The difference shown is not consistent across all processes of a full file listing. In many cases the expected USER value is reflected in the output, but not always.

The first test in investigating this issue is to determine whether the problem is specific to the volafox open files module implementation. Since this part of the open files methodology is taken from the `proc_info` command, the same erroneous output for a given process is expected from both commands. Figure 27 shows that both the `lsof` and `proc_info` commands agree on the USER 'root', demonstrating the problem is *not* isolated to the open files module.

```
$ ./volafox.py –i 10.6.8x86.vmem –o proc_info
list_entry_next   pid   ppid  process name      username
02f74d20            0   0     kernel_task
02f747e0            1   0     launchd           6ad
02f74540           10   1     kextd             root
02f74000           11   1     notifyd           root
03271d20           12   1     diskarbitrationd  root
03271540           15   1     distnoted         root
...
```

Figure 27. volafox `proc_info` output.

The second test confirms correctness of the volafox `proc_info` command implementation. Volafox parses the username based on Suiche's original analysis, which states a "[p]ointer to the process group, `pgrp` structure, allows us to retrieve the username of the person who launched the program because this structure contains a pointer to a structure called `session` with the username" (2010). Figure 28, a screenshot from his conference slides, shows the username `nfi` for the `launchd` process. Here `nfi` (short for Netherlands Forensic Institute) is a local user and therefore cannot be the correct username for the `launchd` process. As Singh explains, "user-level startup is initiated when the kernel executes `/sbin/launchd` as the first user process"



task#	pid	parent pid	name	username
0	0	0	kernel_task	
1	1	0	launchd	nfi
2	10	1	kextd	root
3	11	1	notifyd	root
4	12	1	syslogd	root
5	16	1	update	root
6	19	1	securityd	root
7	21	1	mds	root
8	22	1	mDNSResponder	
9	23	1	loginwindow	nfi
10	24	1	KernelEventAgent	root
11	26	1	hidd	root
12	27	1	fseventsd	root
13	28	1	dynamic_pager	root
14	31	1	diskarbitrationd	root
15	32	1	DirectoryService	root
16	34	1	configd	root
17	37	1	autofsd	root
18	38	1	socketfilterfw	root
19	41	1	distnoted	root
20	47	1	coreservicesd	nfi
21	48	1	WindowServer	root
22	65	1	coreaudiod	nfi
23	69	1	launchd	nfi
24	76	69	Spotlight	nfi
25	77	69	UserEventAgent	nfi
26	79	69	pboard	nfi
27	80	69	Dock	nfi
28	81	69	SystemUIServer	nfi
29	82	69	Finder	nfi
30	83	69	ATSServer	nfi
31	95	69	Terminal	nfi
32	96	95	↳♥:§	
33	97	96	bash	nfi
34	158	69	Xcode	nfi
35	853	80	DashboardClient	nfi
36	1134	69	Property List Ed	nfi
37	2578	69	Safari	nfi
38	2588	95	login	nfi
39	2589	2588	bash	nfi
40	3714	1	ntpd	nfi
41	3837	1	mdworker	nobody
42	3898	1	mdworker	nfi
43	3906	69	AppleSpell	
44	3908	97	dd	nfi

Figure 28. Suiche process list output (2010).

(2006b). The `launchd` process is therefore always associated with the username `root` since the kernel is responsible for executing it. This demonstrates the problem is also not specific to the volafox implementation of Suiche's methodology.

The third test determines whether the discrepancy could be the result of a semantic difference between the user/username fields from the Suiche analysis and the `lsof` manpage. The manpage's "login name of the user to whom the process belongs" could contradict Suiche's definition of a user as "the person who launched the program." The `ps` command offers a variety of keywords associated with users and names that can help. Figure 29 shows the `ps` output for PID 15, the same example previously shown.

```
# ps -p 15 -o ucomm,pid,logname,ruser,user
UCOMM       PID   LOGIN       RUSER       USER
distnoted   15    daemon      daemon      daemon
```

Figure 29. `ps` name keywords.

This result rules out the possibility that Suiche was discussing a different, but nevertheless related and valid username. Figure 30 is the same output for `launchd`, showing the username `nfi` for PID 1 must also be incorrect for all user keywords.

```
# ps -p 1 -o ucomm,pid,logname,ruser,user
UCOMM       PID   LOGIN       RUSER       USER
launchd     1     root        root        root
```

Figure 30. `launchd` name keywords.

In a fourth test, output of the `ps` command is used to analyze the structures involved with username output. The `ps` manpage states the keyword `logname` reports the "login name of user who started the session" and `sess` is the "session ID." Both keywords are likely related to the `session` structure Suiche pulls the username from. Figure 31 shows the output of both for the ongoing example.

```
# ps -p 15 -o ucomm,pid,user,logname,sess
UCOMM        PID   USER         LOGIN        SESS
distnoted    15    daemon       daemon       2fd65f0
```

Figure 31. `ps` session keywords.

The hex output for field "session ID" is equivalent to 50161136 in decimal, so the value appears to be a pointer rather than an integer as one might expect from the `ps` definition. The simple modification in Figure 32 to the `class Session` initializer allows volafox to print this address for comparison as shown in the output Figure 33.

```
class Session(Struct):

  TEMPLATES = {...}

  def __init__(self, addr):
    super(Session, self).__init__(addr)
    print "Session Address: %x" %addr

  ...
}
```

Figure 32. `class Session` testing modification.

```
$ python volafox.py -i 10.6.8x86.vmem -o lsof -p 15
Session address: 2fd65f0
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
distnoted  15 root  cwd    DIR   14,2     1088    2 /dev/null
...
```

Figure 33. `struct session` address.

Because the `ps` keyword `sess` and volafox both report the same address for `struct session`, there is evidence to suggest the `session.s_login[MAXLOGNAME]` member interrogated by volafox is correct for the keyword `logname`. The discrepancy therefore does not appear to be a fault in the source analysis by Suiche.

The final test in this investigation determines if the intermittent discrepancy can be explained by changes to the structure linked-lists *during* memory capture. Hay and Nance (2009) discuss problems associated with inconsistent memory snapshots resulting from state changes during capture. The goal is to show that the `pgrp` and `session` structures that are returned for these failure cases are in fact correct for the process, and not in some momentary transitional state recorded in the image. To accomplish this, volafox is modified to print the values of additional struct members not required for the open files module implementation. The results of these experiments have shown:

1. In all cases observed, **`proc.`**`p_pid == ` **`proc.`**`p_pgpid`, meaning the process group identifier can be used as a reference to a process. Note that as verified in `ps`, the keyword `gid` ≠ `pgid`. The process group identifier references a specific process within a group, it does not identify the group itself.

2. For all failure cases **`proc.`**`p_pgrpid` == **`pgrp.`**`pg_id` == **`session.`**`s_id`. Since all three structures store a correct identifier referencing the source process, a reference error does not appear to be at fault.

3. The **`session.`**`s_leader` member is a `proc` structure pointer described in the comments as "Session leader. (static)". In all failure cases, this points back the source process so there is no reference error apparent from either direction.

4. The **`proc.`**`si_uid` member is the only example in any of the three structures discussed that is typed `uid_t` (user identifier). However, it does *not* correctly identify the process UID output of `ps`. Therefore, supplying the UID rather than username does not appear to be a valid alternative.

Despite the previous analysis, neither the source nor the solution to the username field-reporting error is identified. There is also no known method to determine when the `session` structure returns the correct value. The bug derives from prior work that is not the direct focus of the research and therefore is not explored further.

3.5.2 Sizing the `/dev` Directory.

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 34 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 34. `/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. Table 4 gives 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.

3.6 Summary

Section 3.1 of this chapter introduces the goals for a new volafox module to list open files from a raw memory dump. Section 3.2 covers design recovery of the kernel structures responsible for the process file descriptor table and memory-mapped files. Section 3.3 reviews technical details for the existing volafox source code. Section 3.4 describes the open files module implementation, including the new source file `lsof.py` and external programs `offsets.c` and `printstructs.py`. Finally, Section 3.5 offers an analysis of open issues related to the implementation.

Significant effort is put forth to ensure a well-engineered solution which adheres to fundamental software design principles. The result is a modular, object-oriented implementation with a minimal interface to the existing volafox source. For comparison, the analogous volafox output for the information printed by the UNIX `lsof` command in Figure 3 concludes this chapter as Figure 35.

```
$ ./volafox.py -i 10.6.8x86.vmem -o 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     0t400    611 /dev/ttys000
bash     109  6ad    1u  CHR    16,0     0t400    611 /dev/ttys000
bash     109  6ad    2u  CHR    16,0     0t400    611 /dev/ttys000
bash     109  6ad  255u  CHR    16,0     0t400    611 /dev/ttys000
```

Figure 35. volafox open files listing.

IV. Results and Analysis

This chapter tests the effectiveness of the volafox module implemented for listing open files from an OS X memory capture. As discussed in Section 3.1.2, one reason for selecting the UNIX `lsof` (list open files) command as the output format model for the volafox handles module is the relative ease with which it can be validated. When executed with administrator privileges on a test system, output provided by `lsof` acts as a baseline against which analysis on memory captured from the same system can be measured.

The following sections define a successful research outcome and describe the tools, processes, and definitions used to analyze the implementation. A suite of software test cases consisting of two OS X versions running both 32 and 64-bit kernel architecture is used to validate the tool. Finally, physical memory captured on a variety of Mac models is used to exercise the tool on real-world data.

4.1 Module Evaluation Methodology

A successful implementation of the volafox open files module must accurately report all file handles, adjusted for stated constraints and known deficiencies. However, because the module represents novel research, no tool exists to validate the output using only the image of physical memory. Therefore, validation must compare data that *approximates* the state of open files when the collection occurred. The UNIX `lsof` command offers a source of data for comparison when output is redirected to a file just before suspending a VM or executing the Mac Memory Reader capture tool.

Given the difficulty of direct validation, success must be redefined in terms of the degree to which the volafox output matches that of the UNIX `lsof` command. 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 (Hay & Nance, 2009). Some allowance is necessary to account for changes 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.

By this definition, limitations of the program that are also shared by `lsof` are not an indication of correctness. For example, while the HFS+ filesystem supports 16-bit Unicode file names, `lsof` "only outputs printable […] 8 bit characters" per its manpage. Therefore, the lack of Unicode support within the volafox open files module is not considered a shortcoming of the implementation for the purpose of this research analysis.

4.1.1 Test Configuration and Design.

The following sections provide a step-by-step description of the processes used to configure, collect, and compare the results discussed in Section 4.2 and employ the research tools `capture.py` and `validate.py` developed for analysis.

4.1.1.1 Controlled Test Configuration.

This section describes the process for configuring OS X virtual machines used to validate the research implementation. Output of the process is a virtual machine prepared for the data collection process. The resulting configuration is intended to minimize operating system activity during the time between when the validation data is gathered

and the memory image is created. This is done so that analysis may focus on differences caused by the implementation, rather than those resulting from temporal changes that occur during normal operating conditions. Graphical navigation paths begin with references to items in the OS X menu bar.

1. Install guest using the VMware Fusion Virtual Machine Assistant with default settings for the OS X version being configured. Wizard will select minimum RAM configuration for that version. *Note*: Apple specifies 2 GB of RAM in the system requirements for 10.6 Server, however VMware selects 1 GB which installed and tested without issue during this research.
2. Install guest updates (10.6.8 and 10.7.3 test cases *only*): → Software Update
3. Disable optional virtual hardware and host interaction in VMware Fusion.
    a. Do *not* install the VMware Tools daemon.
    b. Virtual Machine → Settings → USB & Bluetooth → *uncheck all*
    c. Virtual Machine → Settings → Sound Card → Sound Card: *OFF*
    d. Virtual Machine → Settings → CD/DVD → Enable CD/DVD Drive: *OFF*
    e. Virtual Machine → Settings → Display → Accelerate 3D Graphics: *OFF*
    f. Virtual Machine → Settings → Sharing → Shared Folders: *OFF*
4. Disable networking.
    a. Guest: → System Preferences → Network: *remove all interfaces*
    b. VMware Fusion: Virtual Machine → Settings → Network Adaptor → Enable Network Adaptor: *OFF*
5. Remove OS X startup items in the guest.
    a. Select and delete any items in the following directories:

```
~/Library/LaunchAgents
/Library/LaunchAgents
/Library/LaunchDaemons
/Library/StartupItems
```

    b. → System Preferences → Users & Groups → Login Items: *remove all*

6. Disable other daemons in the guest.

   a. → System Preferences → Software Update → Check for updates: *uncheck*

   b. → System Preferences → Date & Time → Set date and time automatically: *unckeck*

   c. → System Preferences → Desktop & Screen Saver → Start screen saver: *never*

   d. → System Preferences → Energy Saver → Computer sleep: *never*; Display sleep: *never*; *uncheck all other options*

   e. Spotlight indexing, Terminal.app:

   ```
   # mdutil -a -i off
   ```

   f. Server administrative daemon (10.6 Server test case *only*), Terminal.app:

   ```
   # cd /System/Library/LaunchDaemons/
   # launchctl unload -w com.apple.servermgrd.plist
   ```

4.1.1.2 Controlled Data Collection.

This section describes the process for collecting validation data and an image of physical memory from virtual machines configured to test the tool. The process outputs a text file containing the handles reported by the `lsof` command, and an image of physical memory saved by VMware Fusion when the virtual machine is suspended (Section 2.4.1). All steps reference actions in the guest OS unless stated otherwise.

1. Launch Terminal.app from `/Applications/Utilities`. Perform an execution of `lsof` with administrator privileges prior to data collection. The `lsof` manpage indicates certain data structures may be cached so this ensures minimum filesystem interaction when the command is run for data collection.

2. Let the system stabilize with no user interaction for approximately 5 minutes.

3. Collect validation data, from Terminal.app:

   ```
   # lsof > ~/lsof.out
   ```

4. Once command-prompt reappears, user immediately navigates to and clicks the suspend button in the upper-left-hand corner of the virtual machine window.

5. Host: copy suspended memory image, from Terminal.app:

   ```
   $ cp PATH/TEST_CASE.vmwarevm/*.vmem ~/Desktop
   ```

6. Resume virtual machine and recover the `lsof.out` validation data file. Because networking and sharing are disabled, USB support is enabled in the guest so the file can be copied to the host via external media.

4.1.1.3 Real-word Data Collection.

This section describes the process used to collect real-world data from Mac computers. The process outputs a text file containing the handles reported by the `lsof` command, and an image of physical memory originally created by the Mac Memory Reader tool, which is then converted using the volafox utility described in Section 3.3.1.

1. Create a collection toolkit using a forensic workstation. First format external USB media as HFS+ in the OS X Disk Utility application. Next, copy the script `capture.py` (Section 4.2.4) and a directory containing Mac Memory Reader kernel extension along with its dependencies to the root of the formatted device.

2. Plug toolkit into the Mac targeted for collection.

3. Double-click the USB device; this should appear on the desktop once mounted. Within the resulting window, double-click the icon named `capture` (the OS X Finder dies not by default show any file extension).

4. A Terminal window will launch and prompt the user for an administrator password. Enter the password and follow the remaining instructions. The script will report the total time elapsed during collection when execution has finished. A new, uniquely named directory is created on the USB toolkit containing both the image of physical memory and the output from `lsof` for comparison.

5. Drag the USB toolkit shown on the desktop to the trash to dismount, then unplug the device.

6. Prior to analysis, the Mac Memory Reader image must first be converted to a linear format compatible with volafox. On a forensic workstation, execute the following from the volafox source code directory:

   ```
   $ python flatten.py SOURCE.mmr DEST.flat
   ```

4.1.1.4 Results Comparison.

The tool `validate.py` is developed to compare output of the `lsof` command with the volafox open files listing using Python's extensive library of list and string operations. The result is a custom `diff` program used to build the tables throughout this chapter and its related appendices. This section describes both the process for using the tool, and also the tool's processes for performing comparison. Two input files are required for the results comparison process. First, the baseline file referred to here as `lsof.out` contains an approximate list of open files from the time the memory was collected. Second, a linear memory image referred to here as `image.mem` which can be analyzed by volafox. Output of the process depends on the source data. For controlled test cases, applicable results are analyzed from the tables in Section 4.2.3 and used to validate the tool. Results for real-world test cases are summarized in the tables of Section 4.2.4. However, these are not considered part of the validation methodology.

1. Obtain a list of open files from `image.mem` using volafox, and redirect output to file. Include `-v` switch to print all handles including those with partial support.

   ```
   $ python volafox -i image.mem -vo lsof > vlfx.out
   ```

2. Input files `vlfx.out` and `lsof.out` are both needed to run `validate.py`. The script compares input using a difference taxonomy (Section 4.1.2). The `-s` switch is used when analyzing 10.7 test cases (E1). Execute the following:

   ```
   $ python validate.py [-s] lsof.out vlfx.out
   ```

   The `validate.py` script automates the following sequence of steps:

   a. Read both files from disk, discard the header line, and store the remaining lines in a 2D array split on whitespace. The resulting matrices use a row for each handle and a column for each `lsof` field described in Table 3.

   b. Adjust the list of processes in the `lsof` matrix for E1 and E3 and the list of handles in the `lsof` matrix for E4.

c. Adjust the process list in the volafox matrix for E2 and E3.

d. Align the list of processes in both matrices and report any differences as F2.

e. Detect differences in the `lsof` COMMAND and USER fields (columns 0 and 2 of the matrix), report as F1 and D1 respectively.

f. Align the list of handles (matrix rows) and report any differences as F3.

g. Detect differences in the optional file-mode descriptor from the `lsof` FD field (column 3) and report any differences as F4.

h. Detect differences in the `lsof` TYPE field (column 4) and report differences as F5. Do not consider socket (C1) or symbolic link (E5) handles.

i. Detect differences in the `lsof` DEVICE field (column 5) and report differences as F6. Do not consider FIFO (E6), non-vnode (C2), or non-HFS+/DEVFS (C3) handles.

j. Detect differences in the `lsof` SIZE/OFF field (column 6) and report differences as F7. Do not consider non-vnode (C2) or non-HFS+/DEVFS (C3) handles, nor those corresponding to the `/dev` directory (D2) or the `ttys` file used by `lsof` (E7).

k. Detect differences in the `lsof` NODE field (column 7) and report differences as F8. Do not consider non-vnode (C2) or non-HFS+/DEVFS (C3) handles.

l. Detect differences in the `lsof` NAME field (column 8) and report differences as F8. Do not consider non-vnode (C2) handles.

3. Print the count of processes or handles affected by the constraints (C1-3), deficiencies (D1,2), explained differences (E1-7), and failures (F1-9) described in Section 4.1.2 to STDOUT. These text results are transcribed into the detailed test case results (Tables 15-28) located in Appendix E.

4. Ranges of real-world test case results are summarized in Tables 12 and 13 for auxiliary research analysis. However, these results are *not* considered part of the implementation validation methodology described in the following steps.

5. Controlled test cases 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. A summary of these results is found in Tables 8-11. Note that the username reporting deficiency (D1) is listed only to emphasize the number of handles affected since it is already classified as a known bug.

6. Each failure reported for the controlled test cases is analyzed manually to determine if it is likely to represent a validation accuracy fault, or an unknown implementation problem. This judgment is based on knowledge of OS X internals and the consistency of the failure across test cases.

7. The difference taxonomy described in Section 4.1.2 is developed through the iterative application of this process. Reclassify failures as deficiencies, constraints or explained differences when possible and repeat.

4.1.2 Analysis Taxonomy.

This section identifies differences between the `lsof` command and output from the volafox open files module as reported by `validate.py`. Relationships between classifications are discussed in Section 4.1.2.4. Order listed does not correspond to the order in which the validation script determines differences; see Section 4.1.1.4 for the execution sequence. Table 7 compares various file types and the output fields to summarize which combinations are affected by the differences classified in the following sections. The table is also useful for visualizing the scope of the implementation with regard to file type and impact of the known deficiencies.

Table 7. 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

4.1.2.1 Constraints.

Constraints are defined as differences in output that occur due to system design decisions. As described in Section 3.1.3, the volafox open files module has several limitations with regard to handle type and filesystem tag.

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`.

4.1.2.2 Deficiencies.

Deficiencies are defined as differences in output that occur due to known bugs. As described in Section 3.5, 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.

4.1.2.3 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. However, because `lsof` is defined as the authoritative standard for measurement, its bugs are classified as explained differences rather than deficiencies.

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`.

E2. Memory captured using the Mac Memory Reader tool (see Section 2.4.2) includes evidence of the process `MacMemoryReader` and its dependency `image`, whereas output from the `lsof` command does not.

E3. Data collected using `capture.py` (4.4.3) 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 36 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 36. `lsof` handle duplication.

In all observed cases, the file descriptor '`twd`' (a composite of `cwd` and `txt`) 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 open files 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.

4.1.2.4 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 Results

This section describes the experimental setup used to validate the volafox open files module, summarizes output from the experiments, and provides analysis of the results. Experimental data is partitioned in two sets: controlled test cases collected in the lab and real-world collections from physical Mac hardware. Only the controlled test cases are directly considered in the validation of the tool, but the real-world data is maintained due to a number of ancillary conclusions that can be derived from it.

4.2.1 Parameters.

Parameters specify the experimental configurations used to validate the tool.

4.2.1.1 Common Collection Parameters.

- Development/analysis platform: iMac (27-inch Mid 2011), 10.7.x, 8 GB RAM
- Volafox base version: r52, research build (with open files support): 1.0
- OS X version (with UNIX `lsof` built-in)
- RAM installed

4.2.1.2 Controlled Test Case Parameters.

- VMware Fusion version: 4.1.0
- Memory capture format: *.vmem (VMware frozen memory file, linear format)
- Darwin kernel architecture

4.2.1.3 Real-world Collection Parameters.

- Darwin kernel architecture: i386
- Mac Memory Reader version: 3.0.0
- `capture.py` build: 1.1
- Mac hardware (model specification)

4.2.2 Factors.

Factors specify the subset of parameters discretely varied across experiments.

4.2.2.1 Virtual Machine Configurations.

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: 1 GB
   Note: by default, 10.6 Snow Leopard installs a 32-bit kernel for the client version. OS X Server was therefore used instead to achieve this configuration.

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

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

4.2.2.2 Real-world Machine Configurations.

1. OS X version: 10.6.8
   RAM installed: 3 GB
   Mac Model: MacBook Pro (15-inch Core 2 Duo), S/N code: X6A

2. OS X version: 10.6.8
   RAM installed: 2 GB
   Mac Model: MacBook Air (13-inch, Late 2010), S/N code: DR2

3. OS X version: 10.6.8
   RAM installed: 3 GB
   Mac Model: iMac (20-inch Late 2006), S/N code: VUW

4. OS X version: 10.6.8
   RAM installed: 4 GB
   Mac Model: MacBook Pro (15-inch, Mid 2010), S/N code: AGU

5. OS X version: 10.6.8
   RAM installed: 2 GB
   Mac Model: Mac mini (Early 2006), S/N code: U35

6. OS X version: 10.6.8
   RAM installed: 1 GB
   Mac Model: iMac (24-inch Mid 2007), S/N code: X8A

7. OS X version: 10.6.8
   RAM installed: 4 GB
   Mac Model: iMac (27-inch, Mid 2010), S/N code: DB5

8. OS X version: 10.6.8
   RAM installed: 4 GB
   Mac Model: MacBook Pro (13-inch, Mid 2010), S/N code: ATM

9. OS X version: 10.7.0
   RAM installed: 4 GB
   Mac Model: MacBook (13-inch Early 2008), S/N code: 0P2

10. OS X version: 10.7.2
    RAM installed: 2 GB
    Mac Model: MacBook Air, S/N code: 18X

4.2.3 Controlled Test Cases.

Because this research involves implementation of a novel tool, direct validation of the results is inherently problematic. An approximation using output from `lsof` therefore demonstrates the capabilities of the system developed. The validation method used amounts to a suite of software test cases that either pass or fail. Resulting failures are then addressed individually, or reclassified in the difference taxonomy before the suite is rerun. 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 taxonomy described in Section 4.1.2. Furthermore, the difficulty in defining, acquiring, and replicating *clean* memory captures means no statistical rigor can be applied to the validation. The majority of firm numbers from test results are therefore located in Appendix E to avoid the inference that results were achieved through performance analysis.

As stated in Section 3.1, one design goal for the system implemented is to provide coverage for a breadth of operating system 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 virtual machine. 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. Section 4.1.1.1 describes steps taken to minimize operating system interference with the state of open files during collection.

Table 8 shows a summary of the results provided in Appendix E (Table 15) for the first controlled test case. After accounting for the constraints, deficiencies, and

Table 8. OS X 10.6.8 x86 test case summary.

Diff	Field	Δ Count	Details
F1	COMMAND	0	
F2	PID	0	
D1	USER	5	15% of usernames misreported
F3	FD	0	
F4	mode	0	
F5	TYPE	0	
F6	DEVICE	0	
F7	SIZE/OFF	1	process: Terminal file: /dev/ptmx lsof: 0t421 volafox: 0t452 diff: +31 bytes
F8	NODE	2	process: notifyd file: /usr/share/zoneinfo/America/New_York lsof: 266103 volafox: 266579 process: notifyd file: /usr/share/zoneinfo/UTC lsof: 226396 volafox: 266606
F9	NAME	0	

explained differences listed in Section 4.1.2 (not shown), this table indicates how similar the volafox open files output is to the `lsof` approximation.

Table 8 indicates the username deficiency (D1) affects a substantial number of processes reported. 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. The node identification failures (F8) are both files related to time zone opened by the `notifyd` process. It is unclear why the notification server would make changes to these files during `lsof` execution, however the behavior has been observed in all controlled test cases.

Table 9 shows a summary for the second controlled test case (Appendix E, Table 16). The offset and node identification failures (F7, F8) are the same process/file combinations described for the 10.6.8 x86 results.

Table 9. OS X 10.6.0 Server x64 test case summary.

Diff	Field	Δ Count	Details
F1	COMMAND	0	
F2	PID	0	
D1	USER	6	17% of usernames misreported
F3	FD	0	
F4	mode	0	
F5	TYPE	0	
F6	DEVICE	0	
F7	SIZE/OFF	1	process: Terminal file: /dev/ptmx lsof: 0t343 volafox: 0t360 diff: +17 bytes
F8	NODE	1	process: notifyd file: /usr/share/zoneinfo/UTC lsof: 92583 volafox: 92789
F9	NAME	0	

Table 10. OS X 10.7.3 x86 test case summary.

Diff	Field	Δ Count	Details
F1	COMMAND	0	
F2	PID	+1	process: launchdadd volafox: 146
D1	USER	17	38 % of usernames misreported
F3	FD	+3	process: launchd volafox: 8 type: SOCKET
			process: launchd volafox: 46 type: SOCKET
			process: launchd volafox: 82 type: PIPE
F4	mode	0	
F5	TYPE	0	
F6	DEVICE	0	
F7	SIZE/OFF	1	process: Terminal file: /dev/ptmx lsof: 0t446 volafox: 0t508 diff: +62 bytes
F8	NODE	1	process: notifyd file: /usr/share/zoneinfo/UTC lsof: 86138 volafox: 86347
F9	NAME	0	

Table 10 shows a summary for the third controlled test case (Appendix E, Table 17). The extra process in the volafox output (F2) is a daemon with the highest PID in the process list. It therefore appears to have been launched after executing `lsof`. Additional file descriptors in the volafox output (F3) belong to `launchd`. Because the `launchd` process manages all other daemons (Singh, 2006b, p. 38), it is very active and therefore volatile. For both 10.7.x test cases the `lsof` and `launchd` processes appear to be confounded. The offset and node identification failures (F7, F8) are the same process/file combinations previously described.

Table 11. OS X 10.7.0 x64 test case summary.

Diff	Field	Δ Count	Details
F1	COMMAND	0	
F2	PID	+1	process: launchdadd volafox: 147
D1	USER	8	19 % of usernames misreported
F3	FD	+4	process: launchd volafox: 8 type: SOCKET
			process: launchd volafox: 79 type: SOCKET
			process: launchd volafox: 85 type: PIPE
			process: WindowServer volafox: txt type: REG **device: -6, size: -7, node: -8, name: -9/0åÜ**
F4	mode	0	
F5	TYPE	0	
F6	DEVICE	0	
F7	SIZE/OFF	1	process: Terminal file: /dev/ptmx lsof: 0t455 volafox: 0t509 diff: +54 bytes
F8	NODE	1	process: notifyd file: /usr/share/zoneinfo/UTC lsof: 86138 volafox: 86347
F9	NAME	0	

Table 11 shows a summary for the last controlled test case (Appendix E, Table 18). Offset and node identification failures (F7, F8), extra process (F2), and the extra `launchd` handles (F3) are the same previously discussed. The extra `WindowServer` file descriptor 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. The current tool has no ability to detect this condition, though the template test method described in Section 3.4.4 could possibly be used if a reliable test case could be determined for the failure. 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.

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 OS X 10.6.x Snow Leopard and 10.7.x Lion operating systems. Third, as described in Section 3.5.1, 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 definition in Section 4.1.

4.2.4 Real-world Data Analysis.

In addition to the controlled test cases described in Section 4.2.2, the volafox open files module was also tested against a set of memory collected from physical machines. The script `capture.py` was developed to automate the collection of memory using the Mac Memory Reader tool described in Section 2.4.2 and a variety of incident response data from Section 2.2.2 for comparison. These real-world collections are invaluable for program debugging and revealing edge cases in the open files 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 Mac Memory Reader (see Appendix F) assures that output from `lsof` is always stale when compared to a dump of physical memory. Finally, the real-world data available does not cover the breadth of OS versions and kernel architectures.

Appendix F summarizes the collections acquired for analysis. As discussed in Section 2.5.2, revision 52 of the volafox project on which the open files implementation is based does support the Mac Memory Reader output format directly. As a result, only i386 captures can be analyzed with the volafox tool and only after conversion to linear format using the `flatten.py` utility. The ten qualifying captures are listed in Section 4.2.2 and the full open files results for each is available in Appendix E.

Table 12. OS X 10.6.8 combined real-world results (8 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	16-51% of usernames misreported
D2	`/dev` directory cannot be sized	0 handles affected
E1	`lsof` process is not shared	1 process removed
E2	`MacMemoryReader` and `image` processes are not shared	2 processes removed
E3	`sh` process is not shared	1 process removed
E4	Duplicate handles labeled FD: '`twd`'	2-5 handles removed
E5	LINK handles are mislabeled	0 handles affected
E6	FIFO handles do not report device identifier	0-2 handles affected
E7	`lsof ttys` file size is not shared	10 handles affected
F1	Δ COMMAND field	0 commands differ
F2	Δ PID field	0-6% 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

Table 12 shows a combined summary of the real-world results for 32-bit samples running OS X 10.6 Snow Leopard (Appendix E, Tables 19-26). 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 6% of processes (F2) and 22% of handles (F3) thrown out for comparison during alignment, `lsof` does *not* approximate the real-world data very closely. This observation supports the earlier statement that such data makes a poor choice for tool validation.

2. The set of non-vnode handles (sockets, pipes, semaphores, etc.) make up a significant portion of the `lsof` results (C2). This observation highlights one opportunity for 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), and `ntfs` (Apple Bootcamp installation of Windows).

4. As concluded previously, high occurrence of the username reporting bug (D1) makes the results of the USER field unreliable. This impacts not only the open files module in volafox, but also affects the process listing module as explained in Section 3.5.1.

5. Explained differences (E1-E7) and the `/dev` sizing deficiency (D2) do not affect a large number of processes and handles. However, they are enumerated in detail because each occurrence is an important data cleaning consideration during validation of the tool.

6. 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. This mirrors the failure behavior observed across test cases in Section 4.2.3.

7. Upon inspection of the failures by hand, 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. Spotlight should therefore be an important consideration during incident response because of this observed impact to the state of open files.

Unlike Snow Leopard, Lion installs a 64-bit kernel for the client OS by default. A 32-bit Lion kernel is only present on older models that received an upgrade installation of Lion from 32-bit Snow Leopard. This results in fewer qualifying samples available for analysis.

Table 13 summarizes the remaining real-world results, both taken from installations of 32-bit 10.7 Lion (Appendix E, Tables 27-28). The samples are from machines running different minor version of the OS and therefore are grouped only for

Table 13. OS X 10.7.x combined real-world results (2 samples).

Diff	Description	Quantity or % Per Sample
C1	SOCKET handles cannot be subtyped	22% of handles affected
C2	Non-vnode handles are not fully supported	35,36% of handles affected
C3	Non-HFS+/DEVFS vnodes are not fully supported	1 handle affected
D1	Δ USER field	40,54% of usernames misreported
D2	`/dev` directory cannot be sized	1 handle affected
E1	`lsof` process is not shared	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`'	5 handles removed
E5	LINK handles are mislabeled	3 handles affected
E6	FIFO handles do not report device identifier	0 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	4,10% of processes removed
F3	Δ FD field	11,14% of handles removed
F4	Δ MODE field	0,2 modes differ
F5	Δ TYPE field	0,2 types differ
F6	Δ DEVICE field	0,1 device identifiers differ
F7	Δ SIZE/OFF field	1% of sizes/offsets differ
F8	Δ NODE field	0,1% of node identifiers differ
F9	Δ NAME field	0,1% of names differ

convenience. Ranges are replaced with series for values that vary since there are only two samples. The general conclusions listed from the 10.6 analysis remain unchanged, but there are a few interesting highlights. First, the 10.7 results include the unsupported filesystem (C3) `mtmfs` used to implement the Mobile Time Machine feature unique to Lion (Siracusa, 2011). Second, 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 conspicuously absent from the volafox output for the 18X model sample (see Section 4.2.2 for model codes).

The real-world data identifies a number of implementation problems that may not have been encountered otherwise. In addition to the truncated process list for 18X, the U35 model sample causes a previous version of the open files module to enter an infinite loop due to a self-referencing process in the linked list. These two cases led to a host of new exception handling code being added to the open files module. Originally for debugging purposes, the new warnings reveal an abundance of broken pointers present throughout the real-world samples. Model X8A for example generated over 1200 warnings alone, though this was an extreme case.

Since the invalid pointer warnings were not observed in the controlled test cases, two explanations are postulated. First, the `flatten.py` image conversion utility may not be trustworthy, or there exists a bug in the Mac Memory Reader capture tool itself. Second, changes to the layout of memory made during capture may result in instability of the linked data structures reflected in the volafox analysis. Given the number of process

and handle changes observed from memory capture to `lsof` execution in the real-world results, the second explanation is more likely. One recommendation to mitigate this problem is to assure memory capture proceeds as rapidly as possible. As indicated in Appendix F, one factor affecting speed is the type of external media used for capture. The results show a 16 Mb/s average increase in capture speed when using an external hard drive over flash storage.

4.3 Summary

Section 4.1 of this chapter introduces the evaluation technique used to validate the results of the volafox open files module by comparison with approximate output from the UNIX `lsof` command. Sections 4.1.1 and 4.1.2 describe the methods used to test the tool and acquire results. Section 4.2 begins with an overview of the experimental parameters and factors used to test the implementation. Section 4.2.3 offers an analysis of controlled test cases used to validate the tool across the breadth of designed operating system and kernel architecture capabilities. Section 4.2.4 closes with an analysis of results from a set of real-world data collections.

This chapter systematically approaches validation of the volafox open files module using a suite of controlled software test cases. The number of failures present after cleaning the data of enumerated constraints, deficiencies, and explained differences is small enough to permit individual analysis of the faults. Most failures can be circumstantially attributed to the precision of the validation method, with one implementation fault outstanding. Because bugs are an expected element of any complex software system, the results of this analysis are deemed an acceptable research outcome.

V. Conclusions and Recommendations

This chapter reviews the overall conclusions provided by the new volafox open files implementation and compares research results with stated objectives. Academic and practical contributions of the work are emphasized. Recommendations for future research topics related to forensic memory analysis on OS X are also provided. Finally, the future of the open source volafox project extended by the implementation is discussed.

5.1 Research Conclusions

This section summarizes research results and the key conclusions derived in Chapter 4. The original research goal and supporting objectives from Section 1.1 are restated first with a response offered for each.

5.1.1 Research Goal.

Implement and document a capability for parsing file handles from raw memory captured on OS X.

Success – the volafox open files module implementation documented in Section 3.4 can list handles associated with regular files, directories, symbolic links, FIFO, and character special files. Memory-mapped files for the process executable are also parsed along with the current working directory. Test results are favorable on raw memory from suspended instances of OS X running in VMware (Section 4.2.3) and memory captured on physical hardware using the Mac Memory Reader tool (Section 4.2.4).

5.1.2 Supporting Objectives.

Perform design recovery of the data structures responsible for handling open files.

Success – the research implementation effectively leverages the data structures and references described in Section 3.2 to find and parse file handle information. Class structure of the open files source file `lsof.py` is organized to closely resemble that of the design recovery used to build the module (Appendix B, D).

Develop a flexible process for programmatically handling structures defined for different kernel architectures and operating system versions. This necessitates extensible software design resilient to changes in future versions of OS X.

Success – the data structure template process described in Sections 3.4.2 – 3.4.3 effectively parses data and pointer members for 18 different classes written for the volafox open files module. Implementation of the process tested favorably against two architectures (i386, x86_64) and two major versions of OS X (10.6 Snow Leopard, 10.7 Lion) in Section 4.2.3. While the template process was found to be flexible across both tested versions of OS X, future extensibility relies on consistent naming of key structure members (Section 3.4.2). Because the development process requires kernel headers defining key structures, extensibility is also dependent on the open source availability of the kernel code.

5.1.3 Additional Research Conclusions.

Testing the implementation also produced several additional conclusions not directly related to the research goal and its supporting objectives. First, the open files module does not reliably output the correct user of a running process, as verified across all controlled and real-world test cases (Section 4.2). No fault could be identified in the implementation, nor any problem with the kernel structure analysis described in prior work (Section 3.5.1). The deficiency is formally listed in Section 4.1.2.2 and represented in the results. Because the login username is only indirectly related to the file handles through the process, this open issue does not affect the overall research outcome.

Second, due to the time required for memory capture on physical hardware, memory analysis tools are difficult to validate on real-world data (Section 4.2.4). Controlled tests using virtual machine memory images should therefore be used in development.

Third, the OS X indexed search technology, Spotlight, changes numerous handles in response to the mounting of external media (Section 4.2.4). This observation should be considered during incident response and subsequent analysis if the memory capture toolkit is delivered via mounted filesystem.

Finally, memory captured on physical hardware suffers from a high number of invalid pointer references, occasionally resulting in malformed linked-lists (Section 4.2.4). Robust exception handling should be implemented to address this problem in a memory analysis tool. Collection must proceed quickly to minimize state changes during memory copy and avoid possible loss of evidence. External hard disk is preferable to flash media due to faster collection speeds (Appendix F).

5.2 Significance of Research

This research makes three primary contributions to the field of forensic memory analysis on OS X.

Design recovery of kernel structures related to open files. While the Darwin kernel foundation of OS X is open source, it is not well documented. Structured memory analysis requires knowledge of kernel organization that must be traced by hand from the source code. The description of data structures and their relationships in this document will be of use to anybody seeking to implement or extend extraction of OS X file handles from raw memory.

Dynamic data structure templates. The repeatable, general-purpose process developed to build structure templates for a range of architecture and operating system configurations can be applied to existing and future volafox modules. It may also prove extensible to future versions of OS X. The method implemented to dynamically select the correct template for a given memory image greatly simplifies the programming logic needed to support multiple configurations.

Volafox open files module implementation. Because the research code was designed around a minimal interface to the existing project source, the new module can be patched against the latest development revision with little or no refactoring. The full source code for the new module is available in Appendix G. With follow on efforts to address outstanding constraints and deficiencies, this module could be a component in a comprehensive tool targeted at technical users and forensic analysts.

5.3 Recommendations for Future Research

The problem outlined in Chapter 1 emphasizes use of forensic memory analysis to replace more invasive incident response methods. Section 2.2.2 lists seven pieces of information that National Institute of Standards and Technology (NIST) specifies should be captured during the volatile collection process. Including this research, volafox now has modules to support four of the seven items. For maximum impact, future OS X memory analysis research efforts should consider the remainder: login sessions, network configuration, and operating system time.

Constraints and deficiencies of the research implementation for listing open files, Sections 4.1.2.1 and 4.1.2.2 respectively, enumerate the work remaining for this specific module. Support for the following features is recommended in particular:

1. *Identifying process ownership* – the inability to consistently determine the user associated with a particular process reduces the credibility of the volafox output and also its usefulness. A process normally expected to run in userland takes on new significance when observed executing as root for example. The bug is pervasive as it affects both the open files and running process modules in volafox so addressing the user problem is a priority.

2. *Support for socket handles* – sockets make up a significant portion of the unsupported handle types. They are of high investigative value and can help identify remote activity on a target or highlight exfiltration of data. Because the capabilities of the volafox network information module are currently limited (Section 2.5.2), the next priority in handle support should include the socket subtypes (Section 4.1.2.1).

3. *Additional filesystem support* – real-world test results identified four filesystems currently unsupported by the open files module: `msdos`, `ntfs`, `cddafs` and `mtmfs`. The `msdos` filesystem (usually FAT32) is a common choice for external media that must be interoperable with OS X and Windows. Though `ntfs` is mounted read-only by OS X, mounts may be present if a Mac is partitioned via Bootcamp to support dual-boot with Windows. Though not observed in testing, two additional network filesystems are also worth considering. The proprietary Apple Filing Protocol (AFP) is the default for OS X filesharing and Common Internet File System (CIFS) is used for shares mounted over Samba.

One final suggestion for future research is to consider the performance analysis of memory capture speed, the interval required to image a target, as it relates to data structure corruption. This research assumes the occurrence of invalid pointers and broken linked data structures (looping references, incomplete listings) is correlated with the time required to image physical memory. Exploring this relationship further might offer important insights with regard to the development of efficient tools and processes for copying physical memory for forensic analysis.

5.3.1 Future of the Volafox Project.

The volafox project currently suffers from a lack of contributing members needed to build a robust forensics capability that can be relied upon by technical users. This thesis represents the second academic work to extend volafox (Leat, 2011), but to move beyond the research domain a larger community is required to develop and validate the tool. An agenda for the upcoming Forensics and Incident Response Summit (SANS Institute, 2012) indicates OS X memory analysis may soon be added to the Volatility framework. Volatility is an established project in the field and the de facto standard for open source forensic memory analysis. Due to the community support and visibility, it is recommended that existing work from volafox be integrated with the future OS X branch of Volatility. Because volafox borrows extensively from the Volatility source, the transition should not require major refactoring.

5.4 Summary

Section 5.1 restates the original research goal and its supporting objectives before comparing both to the conclusions provided in Chapter 4. Testing shows the implementation successfully responds to the research goal and the development process offers a practical demonstration of the derived objectives. Section 5.2 describes the three significant research contributions: design recovery of kernel structures related to open files, a process for generating and selecting dynamic data structure templates, and the volafox open files module itself. Section 5.3 offers recommendations for future research related to OS X forensic memory analysis and the future of the volafox project.

Appendix A. Regular Builds OS X 10.4.4 – 10.7.3

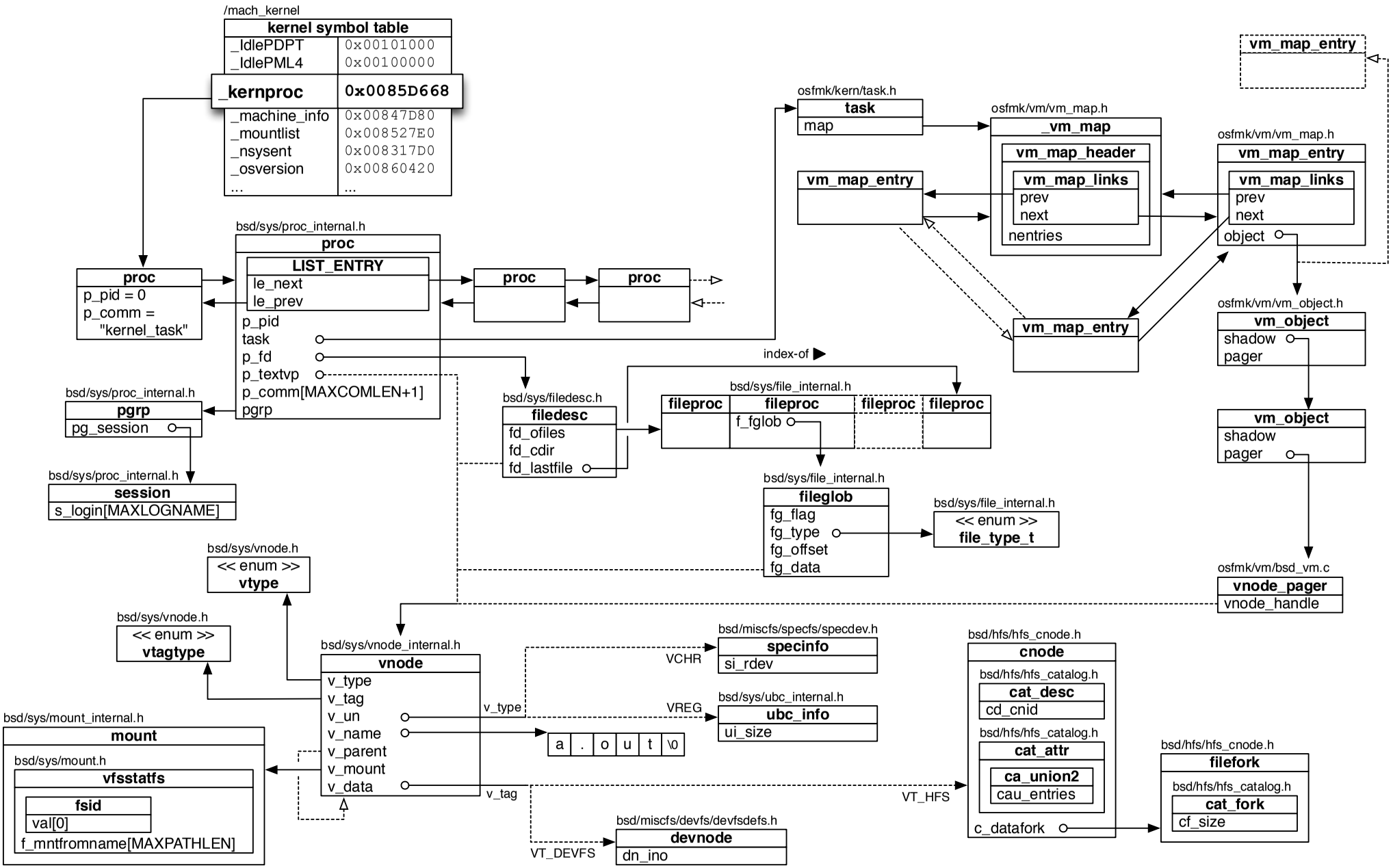
OS X	Build	Date	Darwin	Notes
10.4.4	8G1165	Jan 2006	8.4	http://support.apple.com/kb/HT2343
10.4.5	8G1454	Feb 2006	8.5	http://support.apple.com/kb/HT2581
10.4.6	**8I1119**	Apr 2006	8.6	http://support.apple.com/kb/HT2200
10.4.7	**8J2135a**	Jun 2006	8.7	http://support.apple.com/kb/TA24141
10.4.8	**8L2127**	Sep 2006	8.8	http://support.apple.com/kb/TA24306
10.4.9	**8P2137**	Mar 2007	8.9	http://support.apple.com/kb/HT1525
10.4.10	**8R2232**	Jun 2007	8.10	http://support.apple.com/kb/HT1607
10.4.11	**8S2167**	Nov 2007	8.11	http://support.apple.com/kb/TA24901
10.5	**9A581**	Oct 2007	9.0	Retail DVD release; http://support.apple.com/kb/HT1633
10.5.1	**9B18**	Nov 2007	9.1	http://support.apple.com/kb/HT1566
10.5.2	**9C31**	Feb 2008	9.2	http://support.apple.com/kb/HT1327
10.5.3	**9D34**	May 2008	9.3	http://support.apple.com/kb/HT1141
10.5.4	**9E17**	Jun 2008	9.4	http://support.apple.com/kb/HT1994
	9E25			http://support.apple.com/kb/HT1633
10.5.5	**9F33**	Sep 2008	9.5	http://support.apple.com/kb/HT2405
10.5.6	**9G55**	Dec 2008	9.6	http://support.apple.com/kb/HT3194
	9G66	Jan 2009		Retail DVD release (part of Mac Box Set); http://support.apple.com/kb/HT1633
	9G71			http://support.apple.com/kb/HT3192
10.5.7	**9J61**	May 2009	9.7	http://support.apple.com/kb/HT3397
10.5.8	9L30	Aug 2009	9.8	http://support.apple.com/kb/HT1633
	9L31a			http://support.apple.com/kb/HT3606
	9L34			http://support.apple.com/kb/HT3607
10.6	**10A432**	Aug 2009	10.0	Retail DVD release; http://support.apple.com/kb/HT1633
	10A433			Server retail DVD release; http://support.apple.com/kb/HT1633
10.6.1	**10B504**	Sep 2009	10.1	http://support.apple.com/kb/HT3810
10.6.2	**10C540**	Nov 2009	10.2	http://support.apple.com/kb/HT3874

10.6.3	**10D573**	Mar 2010	10.3	http://support.apple.com/kb/HT4014
	10D575			Retail DVD release; http://support.apple.com/kb/HT1633
	10D578	Apr 2010		http://support.apple.com/kb/DL1017
10.6.4	**10F569**	Jun 2010	10.4	http://support.apple.com/kb/HT4150
	10F616			http://support.apple.com/kb/DL1050
10.6.5	**10H574**	Nov 2010	10.5	http://support.apple.com/kb/HT4250
	10H575			http://support.apple.com/kb/DL1327
10.6.6	**10J567**	Jan 2011	10.6	http://support.apple.com/kb/HT4459
10.6.7	**10J869**	Mar 2011	10.7	http://support.apple.com/kb/HT4472
10.6.8	10K540	Jun 2011	10.8	http://support.apple.com/kb/HT1633
	10K549	Jul 2011		http://support.apple.com/kb/DL1400
10.7	**11A511,a**	Jul 2011	11.0	Retail Mac App Store release; http://support.apple.com/kb/HT1633
	11A511s	Aug 2011		Retail USB drive release
10.7.1	**11B26**	Aug 2011	11.1	http://support.apple.com/kb/HT4764
10.7.2	**11C74**	Oct 2011	11.2	http://support.apple.com/kb/HT4767
10.7.3	**10D50**	Feb 2012	11.3	Releases b and d have been observed, symbol tables appear to be interchangeable; http://support.apple.com/kb/HT5048

Notes:

1. This information largely summarized from Wikipedia release histories for 10.4-10.7, and the Apple knowledgebase article “Finding your Mac OS X version and build information” (http://support.apple.com/kb/HT1633).
2. There are also numerous hardware-specific builds in addition to those listed, some of which are available in the Apple knowledgebase article “Mac OS X versions (builds) for computers” (http://support.apple.com/kb/HT1159).
3. Build numbers in bold are those encountered in the course of this research, and therefore ones for which volafox could be patched to support using the data collected.

Appendix B. Full Structure Diagram



Appendix C. Python `struct` Library

Figure 37 demonstrates the Python standard library function `struct.unpack` as utilized by the open files module `unpacktype()` wrapper. The function is used to convert C structures represented as binary data to a tuple of equivalent Python types. From the documentation http://docs.python.org/library/struct.html:

> `struct.`**`unpack`**`(`*`fmt`*`, `*`string`*`)`¶
> Unpack the string (presumably packed by `pack(`*`fmt`*`, ...)`) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (`len(`*`string`*`)` must equal `calcsize(`*`fmt`*`)`).

Within the context of this research *`string`* is a variable read from file using either `IA32PagedMemoryPae` or `IA32PML4MemoryPae`, whichever object class volafox is using an instance of. Return value of `len(`*`string`*`)` is the number of bytes. The *`fmt`* string literal consists of format characters representing primitive C types from Table 14,

```
STR = 0           # string: char (8-bit) * size
INT = 1           # int:    32 or 64-bit
SHT = 3           # short:  16-bit

def unpacktype(binstr, member, mtype):
  offset = member[1]
  size   = member[2]
  fmt    = ''

  if mtype == STR:
    fmt = str(size) + 's'
  elif mtype == INT:
    fmt = 'I' if size == 4 else 'Q'
  elif mtype == SHT:
    fmt = 'H'

        return struct.unpack(fmt, binstr[offset:size+offset])[0]
```

Figure 37. Simplified `lsof.py unpacktype()` function.

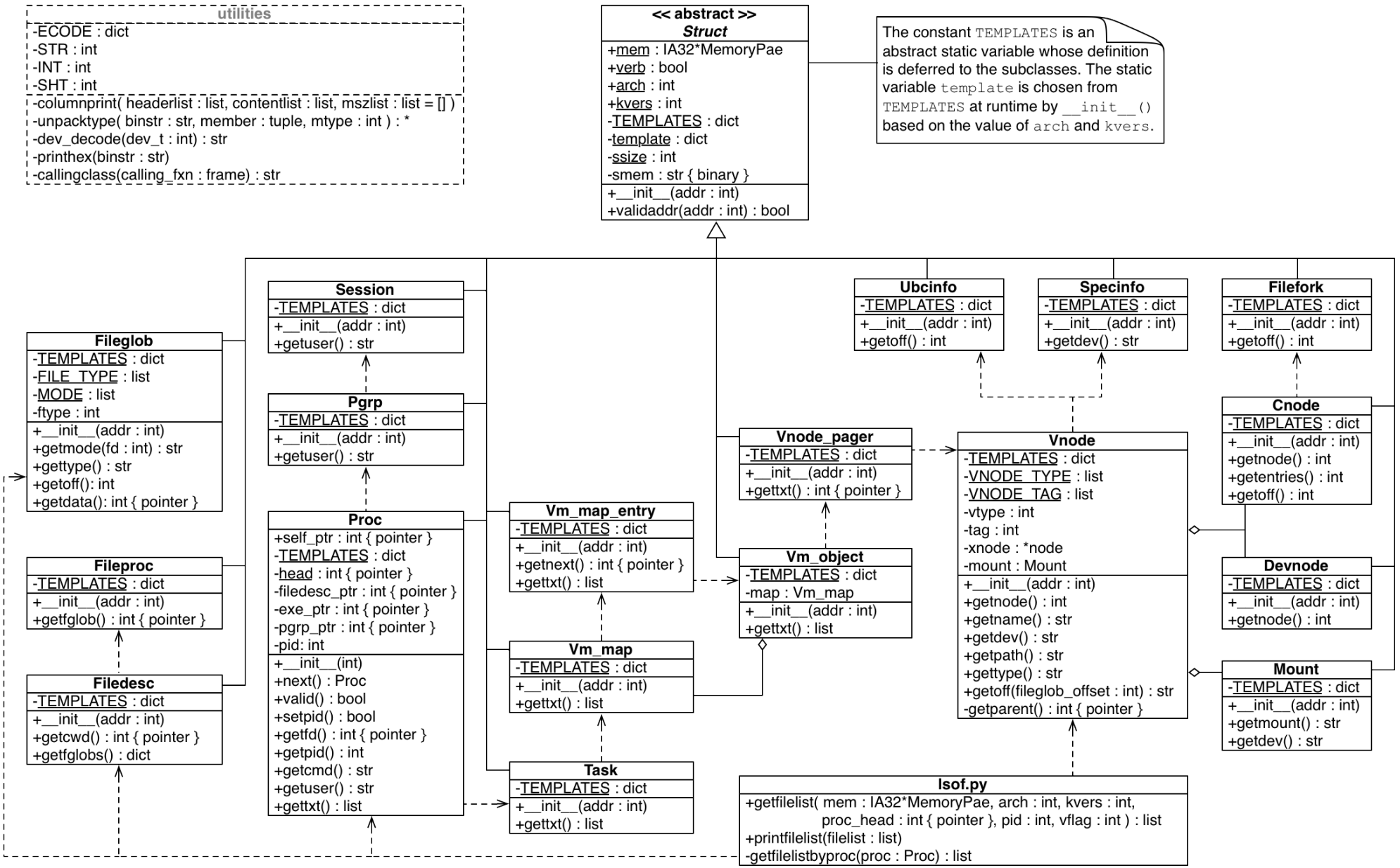
any of which can be preceded by an integral to specify a repeat count (e.g. `'4h'` → `'hhhh'`). Because pointers are not an available type, the format `'I'` or `'Q'` is used to represent a 32 or 64-bit pointer respectively.

The `struct.unpack` function performs endian conversion as needed, but it is not capable of handling the nuances of byte and type alignment for members within a struct. Padding must therefore be explicitly defined in the format string, but bytes marked `'x'` are not included as items in the output tuple. For the format `'2I392xI52sI'` output would consist of the following sequence of types: int, int, int, string, and int.

Table 14. `struct.unpack` format characters.

Format	C Type	Python Type	Standard Size
x	pad byte	no value	
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8
s	char[]	string	
p	char[]	string	

Appendix D. Full UML Diagram



Appendix E. Full Test Results

Table 15. OS X 10.6.8 x86 controlled test case results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	100	624	16.0	handle	F3
C2	TYPE: not(vnode)	239	624	38.3	handle	F3
C3	TYPE: vnode(other)	0	624	0.0	handle	F3
D1	Δ USER	5	34	14.7	process	F2
D2	NAME: /dev	0	624	0.0	handle	F3
E1	COMMAND: lsof,sudo	1 removed			process	
E2	N/A: vm capture					
E3						
E4	FD: twd	1 removed			handle	
E5	TYPE: link	0	624	0.0	handle	F3
E6	TYPE: fifo	0	624	0.0	handle	F3
E7	NODE: node(lsof)	7	624	1.1	handle	F3
F1	Δ COMMAND	0	34	0.0	process	F2
F2	Δ PID	- 0 + 0	34 34	0.0 0.0	process	E1
F3	Δ FD	- 0 + 0	624 624	0.0 0.0	handle	F2, E4 F2
F4	Δ MODE	0	624	0.0	handle	F3
F5	Δ TYPE	0	524	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	385	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	1	378	0.3	handle	F3,C2,C3,D2,E7
F8	Δ NODE	2	385	0.5	handle	F3,C2,C3
F9	Δ NAME	0	385	0.0	handle	F3,C2

Table 16. OS X 10.6.0 Server x64 controlled test case results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	98	642	15.3	handle	F3
C2	TYPE: not(vnode)	241	642	37.5	handle	F3
C3	TYPE: vnode(other)	0	642	0.0	handle	F3
D1	Δ USER	6	35	17.1	process	F2
D2	NAME: /dev	1	642	0.2	handle	F3
E1	COMMAND: lsof,sudo	1 removed			process	
E2	N/A: vm capture					
E3						
E4	FD: twd	1 removed			handle	
E5	TYPE: link	0	642	0.0	handle	F3
E6	TYPE: fifo	0	642	0.0	handle	F3
E7	NODE: node(lsof)	7	642	1.1	handle	F3
F1	Δ COMMAND	0	35	0.0	process	F2
F2	Δ PID	- 0 + 0	35 35	0.0 0.0	process	E1
F3	Δ FD	- 0 + 0	642 642	0.0 0.0	handle	F2, E4 F2
F4	Δ MODE	0	642	0.0	handle	F3
F5	Δ TYPE	0	544	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	401	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	1	393	0.3	handle	F3,C2,C3,D2,E7
F8	Δ NODE	1	401	0.2	handle	F3,C2,C3
F9	Δ NAME	0	401	0.0	handle	F3,C2

Table 17. OS X 10.7.3 x86 controlled test case results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	140	906	15.5	handle	F3
C2	TYPE: not(vnode)	333	906	36.8	handle	F3
C3	TYPE: vnode(other)	0	906	0.0	handle	F3
D1	Δ USER	17	45	37.8	process	F2
D2	NAME: /dev	1	906	0.1	handle	F3
E1	COMMAND: lsof,sudo	2 removed			process	
E2	N/A: vm capture					
E3						
E4	FD: twd	2 removed			handle	
E5	TYPE: link	3	906	0.3	handle	F3
E6	TYPE: fifo	0	906	0.0	handle	F3
E7	NODE: node(lsof)	7	906	0.8	handle	F3
F1	Δ COMMAND	0	45	0.0	process	F2
F2	Δ PID	- 0 + 1	45 46	0.0 2.2	process	E1
F3	Δ FD	- 0 + 3	906 909	0.0 0.3	handle	F2, E4 F2
F4	Δ MODE	0	906	0.0	handle	F3
F5	Δ TYPE	0	763	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	573	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	1	565	0.2	handle	F3,C2,C3,D2,E7
F8	Δ NODE	1	573	0.2	handle	F3,C2,C3
F9	Δ NAME	0	573	0.0	handle	F3,C2

Table 18. OS X 10.7.0 x64 controlled test case results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	133	868	15.3	handle	F3
C2	TYPE: not(vnode)	317	868	36.5	handle	F3
C3	TYPE: vnode(other)	0	868	0.0	handle	F3
D1	Δ USER	8	43	18.6	process	F2
D2	NAME: /dev	1	686	0.1	handle	F3
E1	COMMAND: lsof,sudo	2 removed			process	
E2	N/A: vm capture					
E3						
E4	FD: twd	2 removed			handle	
E5	TYPE: link	3	868	0.3	handle	F3
E6	TYPE: fifo	0	686	0.0	handle	F3
E7	NODE: node(lsof)	7	686	0.8	handle	F3
F1	Δ COMMAND	0	43	0.0	process	F2
F2	Δ PID	- 0 + 1	43 44	0.0 2.3	process	E1
F3	Δ FD	0 4	868 872	0.0 0.5	handle	F2, E4 F2
F4	Δ MODE	0	868	0.0	handle	F3
F5	Δ TYPE	0	732	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	551	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	1	543	0.2	handle	F3,C2,C3,D2,E7
F8	Δ NODE	1	551	0.2	handle	F3,C2,C3
F9	Δ NAME	0	551	0.0	handle	F3,C2

Table 19. OS X 10.6.8, model X6A real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	184	1247	14.8	handle	F3
C2	TYPE: not(vnode)	418	1247	33.5	handle	F3
C3	TYPE: vnode(other)	2	1247	0.2	handle	F3
D1	Δ USER	9	56	16.1	process	F2
D2	NAME: /dev	0	1247	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	3 removed			handle	
E5	TYPE: link	0	1247	0.0	handle	F3
E6	TYPE: fifo	0	1247	0.0	handle	F3
E7	NODE: node(lsof)	10	1247	0.8	handle	F3
F1	Δ COMMAND	0	56	0.0	process	F2
F2	Δ PID	- 0 + 2	56 58	0.0 3.4	process	E1
F3	Δ FD	- 21 + 68	1268 1315	1.7 5.2	handle	F2, E4 F2
F4	Δ MODE	0	1247	0.0	handle	F3
F5	Δ TYPE	0	1063	0.0	handle	F3,C1,E5
F6	Δ DEVICE	2	827	0.2	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	27	817	3.3	handle	F3,C2,C3,D2,E7
F8	Δ NODE	29	827	3.5	handle	F3,C2,C3
F9	Δ NAME	27	829	3.3	handle	F3,C2

Table 20. OS X 10.6.8, model DR2 real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	208	1364	15.2	handle	F3
C2	TYPE: not(vnode)	419	1364	30.7	handle	F3
C3	TYPE: vnode(other)	0	1364	0.0	handle	F3
D1	Δ USER	22	53	41.5	process	F2
D2	NAME: /dev	0	1364	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	4 removed			handle	
E5	TYPE: link	0	1364	0.0	handle	F3
E6	TYPE: fifo	0	1364	0.0	handle	F3
E7	NODE: node(lsof)	10	1364	0.7	handle	F3
F1	Δ COMMAND	0	53	0.0	process	F2
F2	Δ PID	- 0 + 1	53 54	0.0 1.9	process	E1
F3	Δ FD	86 98	1450 1462	5.9 6.7	handle	F2, E4 F2
F4	Δ MODE	1	1364	0.1	handle	F3
F5	Δ TYPE	1	1156	0.1	handle	F3,C1,E5
F6	Δ DEVICE	1	945	0.1	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	93	935	9.9	handle	F3,C2,C3,D2,E7
F8	Δ NODE	74	945	7.8	handle	F3,C2,C3
F9	Δ NAME	20	945	2.1	handle	F3,C2

Table 21. OS X 10.6.8, model VUW real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	186	1065	17.5	handle	F3
C2	TYPE: not(vnode)	369	1065	34.6	handle	F3
C3	TYPE: vnode(other)	37	1065	3.5	handle	F3
D1	Δ USER	22	47	46.8	process	F2
D2	NAME: /dev	0	1065	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	4 removed			handle	
E5	TYPE: link	0	1065	0.0	handle	F3
E6	TYPE: fifo	0	1065	0.0	handle	F3
E7	NODE: node(lsof)	10	1065	0.9	handle	F3
F1	Δ COMMAND	0	47	0.0	process	F2
F2	Δ PID	- 1 + 2	48 49	2.1 4.1	process	E1
F3	Δ FD	- 116 + 146	1181 1211	9.8 12.1	handle	F2, E4 F2
F4	Δ MODE	0	1065	0.0	handle	F3
F5	Δ TYPE	0	879	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	659	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	11	649	1.7	handle	F3,C2,C3,D2,E7
F8	Δ NODE	14	659	2.1	handle	F3,C2,C3
F9	Δ NAME	5	696	0.7	handle	F3,C2

Table 22. OS X 10.6.8, model AGU real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	163	962	16.9	handle	F3
C2	TYPE: not(vnode)	327	962	16.9	handle	F3
C3	TYPE: vnode(other)	0	962	0.0	handle	F3
D1	Δ USER	9	42	21.4	process	F2
D2	NAME: /dev	0	962	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	2 removed			handle	
E5	TYPE: link	0	962	0.0	handle	F3
E6	TYPE: fifo	0	962	0.0	handle	F3
E7	NODE: node(lsof)	10	962	1.0	handle	F3
F1	Δ COMMAND	0	42	0.0	process	F2
F2	Δ PID	- 0 + 2	42 44	0.0 4.5	process	E1
F3	Δ FD	- 34 + 22	996 984	3.4 2.2	handle	F2, E4 F2
F4	Δ MODE	0	962	0.0	handle	F3
F5	Δ TYPE	0	799	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	635	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	2	625	0.3	handle	F3,C2,C3,D2,E7
F8	Δ NODE	2	635	0.3	handle	F3,C2,C3
F9	Δ NAME	0	635	0.0	handle	F3,C2

Table 23. OS X 10.6.8, model U35 real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	361	1972	18.3	handle	F3
C2	TYPE: not(vnode)	595	1972	30.2	handle	F3
C3	TYPE: vnode(other)	18	1972	0.9	handle	F3
D1	Δ USER	24	60	40.0	process	F2
D2	NAME: /dev	0	1972	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	4 removed			handle	
E5	TYPE: link	0	1972	0.0	handle	F3
E6	TYPE: fifo	2	1972	0.1	handle	F3
E7	NODE: node(lsof)	10	1972	0.1	handle	F3
F1	Δ COMMAND	0	60	0.0	process	F2
F2	Δ PID	- 0 + 2	60 62	0.0 3.2	process	E1
F3	Δ FD	- 26 + 51	1998 2023	1.3 2.5	handle	F2, E4 F2
F4	Δ MODE	1	1972	0.1	handle	F3
F5	Δ TYPE	2	1611	0.1	handle	F3,C1,E5
F6	Δ DEVICE	0	1357	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	6	1349	0.4	handle	F3,C2,C3,D2,E7
F8	Δ NODE	2	1359	0.1	handle	F3,C2,C3
F9	Δ NAME	2	1377	0.1	handle	F3,C2

Table 24. OS X 10.6.8, model X8A real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	366	1633	22.4	handle	F3
C2	TYPE: not(vnode)	645	1633	39.5	handle	F3
C3	TYPE: vnode(other)	1	1633	0.1	handle	F3
D1	Δ USER	38	74	51.4	process	F2
D2	NAME: /dev	0	1633	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	5 removed			handle	
E5	TYPE: link	0	1633	0.0	handle	F3
E6	TYPE: fifo	0	1633	0.0	handle	F3
E7	NODE: node(lsof)	10	1633	0.0	handle	F3
F1	Δ COMMAND	0	74	0.0	process	F2
F2	Δ PID	- 1 + 1	75 75	1.3 1.3	process	E1
F3	Δ FD	- 150 + 17	1783 1650	8.4 1.0	handle	F2, E4 F2
F4	Δ MODE	1	1633	0.1	handle	F3
F5	Δ TYPE	0	1267	0.0	handle	F3,C1,E5
F6	Δ DEVICE	1	987	0.1	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	17	977	1.7	handle	F3,C2,C3,D2,E7
F8	Δ NODE	6	987	0.6	handle	F3,C2,C3
F9	Δ NAME	3	988	0.3	handle	F3,C2

Table 25. OS X 10.6.8, model DB5 real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	280	1672	16.7	handle	F3
C2	TYPE: not(vnode)	598	1672	35.8	handle	F3
C3	TYPE: vnode(other)	22	1672	1.3	handle	F3
D1	Δ USER	22	61	36.1	process	F2
D2	NAME: /dev	0	1672	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	2 removed			handle	
E5	TYPE: link	0	1672	0.0	handle	F3
E6	TYPE: fifo	1	1672	0.1	handle	F3
E7	NODE: node(lsof)	10	1672	0.6	handle	F3
F1	Δ COMMAND	0	1672	0.0	process	F2
F2	Δ PID	- 0 + 0	61 61	0.0 0.0	process	E1
F3	Δ FD	- 127 + 50	1799 1722	7.1 2.9	handle	F2, E4 F2
F4	Δ MODE	0	1672	0.0	handle	F3
F5	Δ TYPE	0	1392	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	1051	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	38	1042	3.6	handle	F3,C2,C3,D2,E7
F8	Δ NODE	33	1052	3.1	handle	F3,C2,C3
F9	Δ NAME	11	1074	1.0	handle	F3,C2

Table 26. OS X 10.6.8, model ATM real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	471	2794	16.9	handle	F3
C2	TYPE: not(vnode)	765	2794	27.4	handle	F3
C3	TYPE: vnode(other)	1	2794	0.0	handle	F3
D1	Δ USER	12	65	18.5	process	F2
D2	NAME: /dev	0	2794	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	3 removed			handle	
E5	TYPE: link	0	2794	0.0	handle	F3
E6	TYPE: fifo	0	2794	0.0	handle	F3
E7	NODE: node(lsof)	10	2794	0.0	handle	F3
F1	Δ COMMAND	0	65	0.0	process	F2
F2	Δ PID	- 1 + 3	66 68	1.5 4.4	process	E1
F3	Δ FD	- 352 + 76	3146 2870	11.2 2.6	handle	F2, E4 F2
F4	Δ MODE	2	2794	0.1	handle	F3
F5	Δ TYPE	2	2323	0.1	handle	F3,C1,E5
F6	Δ DEVICE	1	2027	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	55	2017	2.7	handle	F3,C2,C3,D2,E7
F8	Δ NODE	49	2027	2.4	handle	F3,C2,C3
F9	Δ NAME	42	2029	2.1	handle	F3,C2

Table 27. OS X 10.7.0, model 0P2 real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	670	3010	22.3	handle	F3
C2	TYPE: not(vnode)	1087	3010	36.1	handle	F3
C3	TYPE: vnode(other)	1	3010	0.1	handle	F3
D1	Δ USER	36	90	40.0	process	F2
D2	NAME: /dev	1	3010	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	2 removed				
E3	COMMAND: sh	1 removed				
E4	FD: twd	5 removed			handle	
E5	TYPE: link	3	3010	0.1	handle	F3
E6	TYPE: fifo	0	3010	0.0	handle	F3
E7	NODE: node(lsof)	13	3010	0.4	handle	F3
F1	Δ COMMAND	0	90	0.0	process	F2
F2	Δ PID	- 0 + 4	90 94	0.0 4.3	process	E1
F3	Δ FD	- 308 + 139	3318 3149	9.3 4.4	handle	F2, E4 F2
F4	Δ MODE	2	3010	0.1	handle	F3
F5	Δ TYPE	2	2337	0.1	handle	F3,C1,E5
F6	Δ DEVICE	1	1922	0.1	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	27	1908	1.4	handle	F3,C2,C3,D2,E7
F8	Δ NODE	17	1922	0.9	handle	F3,C2,C3
F9	Δ NAME	14	1923	0.7	handle	F3,C2

Table 28. OS X 10.7.2, model 18X real-world results (1 sample).

Diff	Test	Count	Total	%	Unit	Total XRef
C1	TYPE: socket	542	2469	22.0	handle	F3
C2	TYPE: not(vnode)	869	2469	35.2	handle	F3
C3	TYPE: vnode(other)	1	2469	0.0	handle	F3
D1	Δ USER	38	70	54.3	process	F2
D2	NAME: /dev	1	2469	0.0	handle	F3
E1	COMMAND: lsof	1 removed			process	
E2	COMMAND: mmr,image	0 removed				
E3	COMMAND: sh	0 removed				
E4	FD: twd	5 removed			handle	
E5	TYPE: link	3	2469	0.1	handle	F3
E6	TYPE: fifo	0	2469	0.0	handle	F3
E7	NODE: node(lsof)	0	2469	0.0	handle	F3
F1	Δ COMMAND	0	70	0.0	process	F2
F2	Δ PID	- 8 + 0	78 70	10.3 0.0	process	E1
F3	Δ FD	- 290 + 7	2759 2476	10.5 0.3	handle	F2, E4 F2
F4	Δ MODE	0	2469	0.0	handle	F3
F5	Δ TYPE	0	1924	0.0	handle	F3,C1,E5
F6	Δ DEVICE	0	1599	0.0	handle	F3,C2,C3,E6
F7	Δ SIZE/OFF	9	1598	0.6	handle	F3,C2,C3,D2,E7
F8	Δ NODE	5	1599	0.3	handle	F3,C2,C3
F9	Δ NAME	2	1600	0.1	handle	F3,C2

Appendix F. Hardware Capture Summary

#	Arch	OS X	Type	Model	Media	Time	RAM (GB)	MB/s
1	i386	10.6.8	client	AGU	HDD	02:39	4	25.8
2	i386	10.6.8	client	ATM	flash	05:02	4	13.6
3	i386	10.6.8	client	DB5	HDD	02:45	4	24.8
4	i386	10.6.8	client	DR2	flash	02:37	2	13.0
5	i386	10.6.8	client	U35	flash	02:32	2	13.5
6	i386	10.6.8	client	VUW	flash	03:59	3	12.9
7	i386	10.6.8	client	X6A	HDD	02:18	3	22.3
8	i386	10.6.8	client	X8A	HDD	00:38	1	26.9
9	i386	10.7.0	client	0P2	HDD	02:34	4	26.6
10	i386	10.7.2	client	18X	HDD	01:24	2	24.4
11	x86_64	10.6.8	client	M75	HDD	02:22	4	28.8
12	x86_64	10.7.1	server	XYK	HDD	02:32	4	26.9
13	x86_64	10.7.2	client	1G0	HDD	02:03	4	33.3
14	x86_64	10.7.2	client	4R1	HDD	01:01	2	33.6
15	x86_64	10.7.2	client	66D	flash	05:20	4	12.8
16	x86_64	10.7.2	client	66E	HDD	01:55	4	35.6
17	x86_64	10.7.2	client	ATM	HDD	01:55	4	35.6
18	x86_64	10.7.2	client	D6H	HDD	01:56	4	35.3
19	x86_64	10.7.2	client	H2H	flash	05:50	4	11.7
20	x86_64	10.7.2	client	H2H	HDD	02:17	4	29.9
21	x86_64	10.7.2	client	HJP	flash	05:56	4	11.5
22	x86_64	10.7.2	client	HJP	HDD	02:39	4	25.8
23	x86_64	10.7.2	client	HJW	flash	12:47	8	10.7
24	x86_64	10.7.2	client	JWT	HDD	02:42	4	25.3
25	x86_64	10.7.2	client	JWT	HDD	02:42	4	25.3
26	x86_64	10.7.2	client	JWV	flash	07:46	4	8.8
27	x86_64	10.7.2	client	JYC	flash	02:58	2	11.5
28	x86_64	10.7.2	client	JYD	flash	06:36	4	10.3
29	x86_64	10.7.2	client	JYD	HDD	02:38	4	25.9
30	x86_64	10.7.2	client	MGG	HDD	02:39	4	25.8
31	x86_64	10.7.2	client	MW8	flash	05:59	4	11.4
32	x86_64	10.7.2	client	V13	flash	07:07	4	9.59
33	x86_64	10.7.2	client	V7L	HDD	02:52	4	23.8
34	x86_64	10.7.2	client	X92	HDD	01:30	2	22.8
35	x86_64	10.7.2	client	YAM	flash	03:26	2	9.9
36	x86_64	10.7.2	client	ZE2	HDD	03:20	4	20.5
37	x86_64	10.7.2	client	ZE5	HDD	02:43	4	25.1
38	x86_64	10.7.2	server	MFB	flash	08:49	6	11.6
					MIN:	00:38	1	8.8
					MAX:	12:47	8	35.6
							AVG(flash):	11.5
							AVG(HDD):	27.4

Appendix G. Complete Source Code: `lsof.py`

```
#!/usr/bin/env python

'''
      Author: student researcher, osxmem@gmail.com
   Last Edit: 31 Mar 2012
 Description: Research implementation of file handle support for volafox.

   Dependent: x86.py defines read and is_valid_address functions for the
              IA32PagedMemoryPae object passed in the first argument of getfilelist,
              though it is not an import dependency.

Constraints:
  1. NODE field will only be returned for files opened on HFS+ or DEVFS filesystems
  2. Supported filetypes: VNODE
  3. Supported subtypes: REG, DIR, CHR, LINK, FIFO
  4. No unicode support for filenames (8-bit characters only)

Deficiencies:
  1. USER field is not reported correctly for many processes (mismatch with lsof and all
     user-related keywords of ps on the OSX command line)
  3. Files on DEVFS with vnode type of DIR cannot be sized (e.g /dev)

Notes:
  1. All struct classes MUST have at least one element in their template dictionaries
     (even if not fully implemented during development) or there will be serious
     performance issues as the size is sorted out.
'''

import sys
import struct
import inspect

from sys import stderr

# error codes which may be printed in the program output
ECODE = {
     'unsupported': -1,
     'command': -2,
     'pid': -3,
     'fd': -4,
     'type': -5,
     'device': -6,
     'size': -7,
     'node': -8,
     'name': -9
}

####################################### UTILITIES ######################################
```

```
49
50 # convert dev_t (also first member in struct fsid_t) encoding to major/minor device IDs
51 def dev_decode(dev_t):
52
53      # interpreted from the major(x) and minor(x) macros in bsd/sys/types.h
54      maj = (dev_t >> 24) & 255
55      min = dev_t & 16777215
56      return "%d,%d" %(maj, min)
57
58 # print hex representation of a binary string in 8-byte chunks, four to a line
59 def printhex(binstr):
60
61      hexstr = binstr.encode("hex")
62
63      l = len(hexstr)
64      i = 0
65      while i < l:
66           if i+32 < l:
67                line = hexstr[i:i+32]
68           else:
69                line = hexstr[i:]
70           out = ""
71           j = 0
72           for k in xrange(len(line)):
73                out += line[k]
74                if j == 7:
75                     out += ' '
76                     j = 0
77                else:
78                     j += 1
79           print out
80           i += 32
81
82 # print a string matrix as a formatted table of columns
83 def columnprint(headerlist, contentlist, mszlist=[]):
84      num_columns = len(headerlist)
85      size_list   = []
86
87      # start sizing by length of column titles
88      for title in headerlist:
89           size_list.append(len(title))
90
91      # resize based on content
92      for i in xrange(num_columns):
93           for line in contentlist:
94                if len(line) != len(headerlist):
95                     stderr.write("ERROR length of header list does not match content.\n")
96                     return -1
97                if len(line[i]) > size_list[i]:
98                     size_list[i] = len(line[i])
99
100      # check sizing against optional max size list
```

```
101       if len(mszlist) > 0:
102            if len(mszlist) != len(headerlist):
103                 stderr.write("ERROR length of header list does not match max size
list.\n")
104                 return -1
105            for i in xrange(num_columns):
106                 if mszlist[i] < size_list[i] and mszlist[i] > 0:   # -1/0 for unrestricted
sz
107                      if mszlist[i] < len(headerlist[i]):
108                           stderr.write("WARNING max size list and column header length
mismatch.\n")
109                      size_list[i] = mszlist[i]
110
111       # prepend header to content list
112       contentlist = [headerlist] + contentlist
113
114       # build comprehensive, justified, printstring
115       printblock = ""
116       for line in contentlist:
117            printline = ""
118            for i in xrange(num_columns):
119                 if i == 0:
120                      printline += line[i][:size_list[i]].ljust(size_list[i])
121                 elif i == (num_columns-1):
122                      printline += " " + line[i][:size_list[i]]
123                 else:
124                      printline += line[i][:size_list[i]].rjust(size_list[i]+1)
125            printblock += printline + '\n'
126
127       sys.stdout.write('%s' %printblock)
128
129  # mtype (enum)
130  STR = 0         # string: char (8-bit) * size
131  INT = 1         # int:    32 or 64-bit
132  SHT = 3         # short:  16-bit
133
134  # return unpacked member from a struct given its memory and a member template
135  def unpacktype(binstr, member, mtype):
136       offset = member[1]
137       size   = member[2]
138       fmt    = ''
139
140       if mtype == STR:
141            fmt = str(size) + 's'
142       elif mtype == INT:
143            fmt = 'I' if size == 4 else 'Q'
144       elif mtype == SHT:
145            fmt = 'H'
146       else:
147            calling_fxn = sys._getframe(1)
148            stderr.write("ERROR %s.%s tried to unpack the unknown type %d.\n"
%(callingclass(calling_fxn), calling_fxn.f_code.co_name, mtype))
```

```
149             return None
150
151         if struct.calcsize(fmt) != len(binstr[offset:size+offset]):
152             calling_fxn = sys._getframe(1)
153             stderr.write("ERROR %s.%s tried to unpack '%s' (fmt size: %d) from %d bytes.\n"
    %(callingclass(calling_fxn), calling_fxn.f_code.co_name, fmt, struct.calcsize(fmt),
    len(binstr[offset:size+offset])))
154             return None
155
156         return struct.unpack(fmt, binstr[offset:size+offset])[0]
157
158 # return the enclosing class when called inside a function (error reporting)
159 def callingclass(calling_fxn):
160         try:
161             classname = calling_fxn.f_locals['self'].__class__.__name__
162         except KeyError:
163             classname = "<unknown>"
164         return classname
165
166 ################################### PRIVATE CLASSES ####################################
167
168 # parent from which all structures derive, an abstract class
169 class Struct(object):
170
171         # static variables (common to all structure subclasses)
172         mem       = None
173         verb      = False
174         arch      = -1
175         kvers     = -1
176
177         # abstract static variables (subclass-specific)
178         TEMPLATES = None
179         template  = None
180         ssize     = -1
181
182         def validaddr(self, addr):
183             if addr == 0:
184                 calling_fxn = sys._getframe(1)
185                 stderr.write("WARNING %s.%s was passed a NULL address.\n"
    %(callingclass(calling_fxn), calling_fxn.f_code.co_name))
186                 return False
187             elif not(Struct.mem.is_valid_address(addr)):
188                 calling_fxn = sys._getframe(1)
189                 stderr.write("WARNING %s.%s was passed the invalid address %.8x.\n"
    %(callingclass(calling_fxn), calling_fxn.f_code.co_name, addr))
190                 return False
191             return True
192
193         def __init__(self, addr):
194             self.smem = None
195
196             if self.__class__.template == None:
```

```
197
198                 # configure template based on architecture and kernel version
199                 if Struct.arch in self.__class__.TEMPLATES:
200                       if Struct.kvers in self.__class__.TEMPLATES[Struct.arch]:
201                             self.__class__.template =
self.__class__.TEMPLATES[Struct.arch][Struct.kvers]
202                       else:
203                             stderr.write("ERROR %s has no template for x%d Darwin %d.x.\n"
%(self.__class__.__name__, Struct.arch, Struct.kvers))
204                             sys.exit()
205                 else:
206                       stderr.write("ERROR %s does not support %s architecture.\n"
%(self.__class__.__name__, str(Struct.arch)))
207                       sys.exit()
208
209                 # set size of the structure by iterating over template
210                 for item in self.__class__.template.values():
211                       if ( item[1] + item[2] ) > self.__class__.ssize:
212                             self.__class__.ssize = item[1] + item[2]
213
214           if self.validaddr(addr):
215                 self.smem = Struct.mem.read(addr, self.__class__.ssize);
216           else:
217                 stderr.write("ERROR instance of %s failed to construct with address
%.8x.\n" %(self.__class__.__name__, addr))
218
219 # Cnode --> Filefork
220 class Filefork(Struct):
221
222     TEMPLATES = {
223           32:{
224                 10:{'ff_data':('struct
cat_fork',16,96,'',{'cf_size':('off_t',16,8,'SIZE/OFF(LINK)')})}
225                 , 11:{'ff_data':('struct
cat_fork',16,96,'',{'cf_size':('off_t',16,8,'SIZE/OFF(LINK)')})}
226           },
227           64:{
228                 10:{'ff_data':('struct
cat_fork',32,96,'',{'cf_size':('off_t',32,8,'SIZE/OFF(LINK)')})}
229                 , 11:{'ff_data':('struct
cat_fork',32,96,'',{'cf_size':('off_t',32,8,'SIZE/OFF(LINK)')})}
230           }
231     }
232
233     def __init__(self, addr):
234           super(Filefork, self).__init__(addr)
235
236     def getoff(self):
237           return unpacktype(self.smem, self.template['ff_data'][4]['cf_size'], INT)
238
239 # Vnode --> Cnode
240 class Cnode(Struct):
```

```
241
242       TEMPLATES = {
243             32:{
244                   10:{'c_desc':('struct
cat_desc',68,20,'',{'cd_cnid':('cnid_t',80,4,'NODE')}),'c_attr':('struct
cat_attr',88,92,'',{'ca_fileid':('cnid_t',88,4,'NODE'),'ca_union2':('union',140,4,'entries
->SIZE/OFF(dir)')}),'c_datafork':('struct filefork *',204,4,'->datafork')}
245                   , 11:{'c_desc':('struct
cat_desc',72,20,'',{'cd_cnid':('cnid_t',84,4,'NODE')}),'c_attr':('struct
cat_attr',92,92,'',{'ca_fileid':('cnid_t',92,4,'NODE'),'ca_union2':('union',144,4,'entries
->SIZE/OFF(dir)')}),'c_datafork':('struct filefork *',208,4,'->datafork')}
246             },
247             64:{
248                   10:{'c_desc':('struct
cat_desc',104,24,'',{'cd_cnid':('cnid_t',116,4,'NODE')}),'c_attr':('struct
cat_attr',128,120,'',{'ca_fileid':('cnid_t',128,4,'NODE'),'ca_union2':('union',204,4,'entr
ies->SIZE/OFF(dir)')}),'c_datafork':('struct filefork *',288,8,'->datafork')}
249                   , 11:{'c_desc':('struct
cat_desc',112,24,'',{'cd_cnid':('cnid_t',124,4,'NODE')}),'c_attr':('struct
cat_attr',136,120,'',{'ca_fileid':('cnid_t',136,4,'NODE'),'ca_union2':('union',212,4,'entr
ies->SIZE/OFF(dir)')}),'c_datafork':('struct filefork *',296,8,'->datafork')}
250             }
251       }
252
253       def __init__(self, addr):
254             super(Cnode, self).__init__(addr)
255
256       def getnode(self):
257             return unpacktype(self.smem, self.template['c_desc'][4]['cd_cnid'], INT)
258
259       def getentries(self):          # used to calculate size for DIR files
260             return unpacktype(self.smem, self.template['c_attr'][4]['ca_union2'], INT)
261
262       def getoff(self):              # returns the size for LINK files
263             datafork_ptr = unpacktype(self.smem, self.template['c_datafork'], INT)
264             datafork = Filefork(datafork_ptr)
265             return datafork.getoff()
266
267 # Vnode --> Devnode
268 class Devnode(Struct):
269
270       TEMPLATES = {
271             32:{
272                   10:{'dn_ino':('ino_t',112,4,'NODE(CHR)')}
273                   , 11:{'dn_ino':('ino_t',112,4,'NODE(CHR)')}
274             },
275             64:{
276                   10:{'dn_ino':('ino_t',192,8,'NODE(CHR)')}
277                   , 11:{'dn_ino':('ino_t',192,8,'NODE(CHR)')}
278             }
279       }
280
```

```
281       def __init__(self, addr):
282             super(Devnode, self).__init__(addr)
283
284       def getnode(self):
285             return unpacktype(self.smem, self.template['dn_ino'], INT)
286
287 # Vnode --> Specinfo
288 class Specinfo(Struct):
289
290       TEMPLATES = {
291             32:{
292                   10:{'si_rdev':('dev_t',12,4,'->DEVICE(CHR)')}
293                   , 11:{'si_rdev':('dev_t',12,4,'->DEVICE(CHR)')}
294             },
295             64:{
296                   10:{'si_rdev':('dev_t',24,4,'->DEVICE(CHR)')}
297                   , 11:{'si_rdev':('dev_t',24,4,'->DEVICE(CHR)')}
298             }
299       }
300
301       def __init__(self, addr):
302             super(Specinfo, self).__init__(addr)
303
304       def getdev(self):
305             dev_t = unpacktype(self.smem, self.template['si_rdev'], INT)
306             return dev_decode(dev_t)
307
308 # Vnode --> Ubcinfo
309 class Ubcinfo(Struct):
310
311       TEMPLATES = {
312             32:{
313                   10:{'ui_size':('off_t',20,8,'SIZE/OFF(REG)')}
314                   , 11:{'ui_size':('off_t',20,8,'SIZE/OFF(REG)')}
315             },
316             64:{ # NOTE: 10.6/7x64 offset for ui_size edited manually 32 --> 40
317                   10:{'ui_size':('off_t',40,8,'SIZE/OFF(REG)')}
318                   , 11:{'ui_size':('off_t',40,8,'SIZE/OFF(REG)')}
319             }
320       }
321
322       def __init__(self, addr):
323             super(Ubcinfo, self).__init__(addr)
324
325       def getoff(self):
326             return unpacktype(self.smem, self.template['ui_size'], INT)
327
328 # Vnode --> Mount
329 class Mount(Struct):
330
331       TEMPLATES = {
332             32:{
```

```
333                 10:{'mnt_vfsstat':('struct
vfsstatfs',76,2152,'',{'f_fsid':('fsid_t',132,8,'',{'val[0]':('int32_t',132,4,'-
>DEVICE'),'val[1]':('int32_t',136,4,'')}),'f_mntonname':('char[]',168,1024,'->NAME')})}
334                 , 11:{'mnt_vfsstat':('struct
vfsstatfs',76,2152,'',{'f_fsid':('fsid_t',132,8,'',{'val[0]':('int32_t',132,4,'-
>DEVICE'),'val[1]':('int32_t',136,4,'')}),'f_mntonname':('char[]',168,1024,'->NAME')})}
335           },
336           64:{
337                 10:{'mnt_vfsstat':('struct
vfsstatfs',136,2164,'',{'f_fsid':('fsid_t',196,8,'',{'val[0]':('int32_t',196,4,'-
>DEVICE'),'val[1]':('int32_t',200,4,'')}),'f_mntonname':('char[]',232,1024,'->NAME')})}
338                 , 11:{'mnt_vfsstat':('struct
vfsstatfs',132,2164,'',{'f_fsid':('fsid_t',192,8,'',{'val[0]':('int32_t',192,4,'-
>DEVICE'),'val[1]':('int32_t',196,4,'')}),'f_mntonname':('char[]',228,1024,'->NAME')})}
339           }
340     }
341
342     def __init__(self, addr):
343           super(Mount, self).__init__(addr)
344
345     def getmount(self):
346           return unpacktype(self.smem, Mount.template['mnt_vfsstat'][4]['f_mntonname'],
STR).split('\x00', 1)[0].strip('\x00')
347
348     def getdev(self):
349           dev_t = unpacktype(self.smem,
Mount.template['mnt_vfsstat'][4]['f_fsid'][4]['val[0]'], INT)
350           return dev_decode(dev_t)
351
352 # Proc     --> Vnode (exe)
353 # Filesesc --> Vnode (cwd)
354 # Fileglob --> Vnode
355 # Vnode    --> Vnode (parent)
356 class Vnode(Struct):
357
358     TEMPLATES = {
359           32:{
360
     10:{'v_type':('uint16_t',68,2,'TYPE(vnode)'),'v_tag':('uint16_t',70,2,'vfs-
type'),'v_un':('union',76,4,'->ubc_info/specinfo'),'v_name':('const char
*',116,4,'NAME'),'v_parent':('vnode_t',120,4,'-
>vnode(parent)'),'v_mount':('mount_t',136,4,'->mount'),'v_data':('void *',140,4,'-
>cnode/devnode')}
361                 ,
11:{'v_type':('uint16_t',64,2,'TYPE(vnode)'),'v_tag':('uint16_t',66,2,'vfs-
type'),'v_un':('union',72,4,'->ubc_info/specinfo'),'v_name':('const char
*',112,4,'NAME'),'v_parent':('vnode_t',116,4,'-
>vnode(parent)'),'v_mount':('mount_t',132,4,'->mount'),'v_data':('void *',136,4,'-
>cnode/devnode')}
362           },
363           64:{
364
```

```
     10:{'v_type':('uint16_t',112,2,'TYPE(vnode)'),'v_tag':('uint16_t',114,2,'vfs-
type'),'v_un':('union',120,8,'->ubc_info/specinfo'),'v_name':('const char
*',184,8,'NAME'),'v_parent':('vnode_t',192,8,'-
>vnode(parent)'),'v_mount':('mount_t',224,8,'->mount'),'v_data':('void *',232,8,'-
>cnode/devnode')}
365                  ,
11:{'v_type':('uint16_t',104,2,'TYPE(vnode)'),'v_tag':('uint16_t',106,2,'vfs-
type'),'v_un':('union',112,8,'->ubc_info/specinfo'),'v_name':('const char
*',176,8,'NAME'),'v_parent':('vnode_t',184,8,'-
>vnode(parent)'),'v_mount':('mount_t',216,8,'->mount'),'v_data':('void *',224,8,'-
>cnode/devnode')}
366             }
367     }
368
369     # NOTE 1: type LINK below is called just "LNK" in the source but lsof uses "LINK"
370     # NOTE 2: 10.7 version of lsof appears to be broken for LINK types, it outputs the
371     #         undocumented type "0012" instead
372     # NOTE 3: these static lists defined in bsd/sys/vnode.h but modified for printing
373     VNODE_TYPE = ["NON", "REG", "DIR", "BLK", "CHR", "LINK", "SOCK", "FIFO", "BAD",
"STR", "CPLX"]
374     VNODE_TAG = ['NON', 'UFS', 'NFS', 'MFS', 'MSDOSFS', 'LFS', 'LOFS', 'FDESC',
'PORTAL', 'NULL', 'UMAP', 'KERNFS', 'PROCFS', 'AFS', 'ISOFS', 'UNION', 'HFS', 'ZFS',
'DEVFS', 'WEBDAV', 'UDF', 'AFP', 'CDDA', 'CIFS', 'OTHER']
375
376     def __init__(self, addr):
377          super(Vnode, self).__init__(addr)
378          self.vtype = None
379          self.tag   = None
380          self.xnode = None     # cnode, devnode
381          self.mount = None
382
383     def getnode(self):
384
385          if self.xnode == None:
386               x_node_ptr = unpacktype(self.smem, self.template['v_data'], INT)
387
388               if self.tag == None:
389                    self.tag = unpacktype(self.smem, self.template['v_tag'], SHT)
390
391               if self.tag == 16:        # VT_HFS
392                    self.xnode = Cnode(x_node_ptr)
393
394               elif self.tag == 18: # VT_DEVFS
395                    self.xnode = Devnode(x_node_ptr)
396
397               else:
398                    if self.tag < len(Vnode.VNODE_TAG):
399                         s_tag = Vnode.VNODE_TAG[self.tag]
400                    else:
401                         s_tag = str(self.tag)
402                    stderr.write("WARNING Vnode.getnode(): unsupported FS tag %s,
returning %d.\n" %(s_tag, ECODE['node']))
```

```
403                     return ECODE['node']
404
405           return self.xnode.getnode()
406
407      def getname(self):
408           name_ptr = unpacktype(self.smem, self.template['v_name'], INT)
409
410           if name_ptr == 0 or not(Struct.mem.is_valid_address(name_ptr)):
411                return None
412
413           # NOTE: this may be trouble for the 255 UTF-16 filename characters HFS+ allows
414           name_addr = Struct.mem.read(name_ptr, 255)
415           name = struct.unpack('255s', name_addr)[0]
416           return name.split('\x00', 1)[0].strip('\x00')
417
418      def getparent(self):
419           parent_ptr = unpacktype(self.smem, self.template['v_parent'], INT)
420
421           if parent_ptr == 0 or not(Struct.mem.is_valid_address(parent_ptr)):
422                return None
423           return parent_ptr
424
425      def getdev(self):
426
427           if self.tag == None:
428                self.tag = unpacktype(self.smem, self.template['v_tag'], SHT)
429
430           if self.tag == 18:   # CHR
431                vu_specinfo = unpacktype(self.smem, self.template['v_un'], INT)
432
433                # this pointer is invalid for /dev (special case DIR using VT_DEVFS)
434                if not(vu_specinfo == 0) and Struct.mem.is_valid_address(vu_specinfo):
435                     specinfo = Specinfo(vu_specinfo)
436                     return specinfo.getdev()
437
438           # default return for REG/DIR/LINK
439           if self.mount == None:
440                mount_ptr = unpacktype(self.smem, self.template['v_mount'], INT)
441
442                if mount_ptr == 0 or not(Struct.mem.is_valid_address(mount_ptr)):
443                     stderr.write("WARNING Vnode.getdev(): v_mount pointer invalid,
returning %d.\n" %ECODE['device'])
444                     return ECODE['device']
445
446                self.mount = Mount(mount_ptr)
447
448           return self.mount.getdev()
449
450      def getpath(self):
451           path      = ""
452           mntonname = ""
453           parent          = self
```

```
454
455             if self.tag == None:
456                  self.tag = unpacktype(self.smem, self.template['v_tag'], SHT)
457
458             if self.mount == None:
459                  mount_ptr = unpacktype(self.smem, self.template['v_mount'], INT)
460
461                  if mount_ptr == 0 or not(Struct.mem.is_valid_address(mount_ptr)):
462                       stderr.write("WARNING Vnode.getpath(): v_mount pointer invalid,
    returning %d.\n" %ECODE['name'])
463                       mntonname = str(ECODE['name'])
464
465                  else:
466                       self.mount = Mount(mount_ptr)
467
468             if self.mount != None:
469                  mntonname = self.mount.getmount()
470
471             while True:
472                  parent_ptr = parent.getparent()
473                  if parent_ptr == 0 or not(Struct.mem.is_valid_address(parent_ptr)):
474                       break
475
476                  name = parent.getname()
477                  if name == None:
478                       break
479
480                  path = name + "/" + path
481                  parent = Vnode(parent_ptr)
482
483             if len(path) < 2:                          # file is root
484                  return mntonname
485
486             if len(mntonname) == 1:                    # mount is root, delete trailing slash
487                  return mntonname + path[:-1]
488
489             return mntonname + "/" + path[:-1]   # mount + path, delete trailing slash
490
491        def gettype(self):
492
493             if self.vtype == None:
494                  self.vtype = unpacktype(self.smem, self.template['v_type'], SHT)
495
496             if self.vtype < len(Vnode.VNODE_TYPE):
497                  return Vnode.VNODE_TYPE[self.vtype]
498
499             return -1 # check for this in the Vnode_pager validation
500
501        def getoff(self, fileglob_offset):
502
503             if self.vtype == None:
504                  self.vtype = unpacktype(self.smem, self.template['v_type'], SHT)
```

```
505             if self.tag == None:
506                  self.tag = unpacktype(self.smem, self.template['v_tag'], SHT)
507
508             # NOTE: UBC information not valid for vnodes marked as VSYSTEM
509             if self.vtype == 1:            # REG
510                  ubcinfo_ptr = unpacktype(self.smem, self.template['v_un'], INT)
511
512                  if ubcinfo_ptr == 0 or not(Struct.mem.is_valid_address(ubcinfo_ptr)):
513                       stderr.write("WARNING Vnode.getoff(): v_un pointer invalid, returning
%d.\n" %(ECODE['size']))
514                       return ECODE['size']
515
516                  ubcinfo = Ubcinfo(ubcinfo_ptr)
517                  return ubcinfo.getoff()
518
519             elif self.tag == 16:      # VT_HFS
520                  if self.xnode == None:
521                       x_node_ptr = unpacktype(self.smem, self.template['v_data'], INT)
522                       self.xnode = Cnode(x_node_ptr)
523
524                  if self.vtype == 2:       # DIR
525                       entries = self.xnode.getentries()
526                       return (entries + 2) * 34 # AVERAGE_HFSDIRENTRY_SIZE: bsd/hfs/hfs.h
527
528                  elif self.vtype == 5:     # LINK
529                       return self.xnode.getoff()
530
531                  elif self.vtype == 7:     # FIFO
532                       return "0t%i" %fileglob_offset
533
534             elif self.tag == 18:      # VT_DEVFS
535                  if self.vtype == 4:       # CHR
536                       return "0t%i" %fileglob_offset
537
538                  elif self.vtype == 2:     # /dev
539                       return "-1"               # not returning ECODE because this
deficiency known
540
541             if self.tag < len(Vnode.VNODE_TAG):
542                  s_tag = Vnode.VNODE_TAG[self.tag]
543             else:
544                  s_tag = str(self.tag)
545             if self.vtype < len(Vnode.VNODE_TYPE):
546                  s_type = Vnode.VNODE_TYPE[self.vtype]
547             else:
548                  s_type = str(self.vtype)
549             stderr.write("WARNING Vnode.getoff(): unsupported type %s, tag %s. Returning
%d.\n" %(s_type, s_tag, ECODE['size']))
550             return ECODE['size']
551
552 # Fileproc --> Fileglob
553 class Fileglob(Struct):
```

```
554
555       TEMPLATES = {
556            32:{
557
      10:{'fg_flag':('int32_t',16,4,'MODE'),'fg_type':('file_type_t',20,4,'FTYPE'),'fg_off
set':('off_t',40,8,'SIZE/OFF'),'fg_data':('caddr_t',48,4,'->vnode')}
558                 ,
11:{'fg_flag':('int32_t',16,4,'MODE'),'fg_type':('file_type_t',20,4,'FTYPE'),'fg_offset':(
'off_t',40,8,'SIZE/OFF'),'fg_data':('caddr_t',48,4,'->vnode')}
559            },
560            64:{
561
      10:{'fg_flag':('int32_t',32,4,'MODE'),'fg_type':('file_type_t',36,4,'FTYPE'),'fg_off
set':('off_t',64,8,'SIZE/OFF'),'fg_data':('caddr_t',72,8,'->vnode')}
562                 ,
11:{'fg_flag':('int32_t',32,4,'MODE'),'fg_type':('file_type_t',36,4,'FTYPE'),'fg_offset':(
'off_t',64,8,'SIZE/OFF'),'fg_data':('caddr_t',72,8,'->vnode')}
563            }
564       }
565
566       # global defined in bsd/sys/file_internal.h but modified to match lsof output
567       FILE_TYPE = ["-1", "VNODE", "SOCKET", "PSXSHM", "PSXSEM", "KQUEUE", "PIPE",
"FSEVENT"]
568       MODE      = ["  ", "r ", "w ", "u "]
569
570       def __init__(self, addr):
571            super(Fileglob, self).__init__(addr)
572            self.ftype = None
573
574       def getmode(self, fd):
575            self.ftype = unpacktype(self.smem, self.template['fg_type'], INT)
576            filemode = "  "
577
578            # NOTE: in limited lsof testing types known to include file mode reporting are:
579            #       VNODE, SOCKET, PSXSHM, PSXSEM, and KQUEUE. Others do not append any
580            #       character to the FD identifier.
581            if self.ftype in xrange(1,6):
582                 flag = unpacktype(self.smem, self.template['fg_flag'], INT)
583                 filemode = Fileglob.MODE[flag & 3]
584
585            return str(fd)+filemode
586
587       def gettype(self):
588
589            if self.ftype == None:
590                 self.ftype = unpacktype(self.smem, self.template['fg_type'], INT)
591
592            if self.ftype < 0 or self.ftype > ( len(Fileglob.FILE_TYPE) -1 ):
593                 stderr.write("WARNING Fileglob.gettype(): unknown file type %d, excluding
this result.\n" %self.ftype)
594                 return -1      # check for this in the getfilelistbyproc()
595
```

```
596             return Fileglob.FILE_TYPE[self.ftype]
597
598     def getoff(self):
599             return unpacktype(self.smem, self.template['fg_offset'], INT)
600
601     def getdata(self):
602             data_ptr = unpacktype(self.smem, self.template['fg_data'], INT)
603
604             if self.validaddr(data_ptr):
605                     return data_ptr
606             return None
607
608 # Filedesc --> Fileproc
609 class Fileproc(Struct):
610
611     TEMPLATES = {
612             32:{
613                     10:{'f_fglob':('struct fileglob *',8,4,'->fileglob')}
614                     , 11:{'f_fglob':('struct fileglob *',8,4,'->fileglob')}
615             },
616             64:{
617                     10:{'f_fglob':('struct fileglob *',8,8,'->fileglob')}
618                     , 11:{'f_fglob':('struct fileglob *',8,8,'->fileglob')}
619             }
620     }
621
622     def __init__(self, addr):
623             super(Fileproc, self).__init__(addr)
624
625     def getfglob(self):
626             fileglob_ptr = unpacktype(self.smem, self.template['f_fglob'], INT)
627
628             if self.validaddr(fileglob_ptr):
629                     return fileglob_ptr
630             return None
631
632 # Proc --> Filedesc
633 class Filedesc(Struct):
634
635     TEMPLATES = {
636             32:{
637                     10:{'fd_ofiles':('struct fileproc **',0,4,'-
>fileproc[]'),'fd_cdir':('struct vnode *',8,4,'->CWD'),'fd_lastfile':('int',20,4,'-
>fileproc[LAST_INDEX]')}
638                     , 11:{'fd_ofiles':('struct fileproc **',0,4,'-
>fileproc[]'),'fd_cdir':('struct vnode *',8,4,'->CWD'),'fd_lastfile':('int',20,4,'-
>fileproc[LAST_INDEX]')}
639             },
640             64:{
641                     10:{'fd_ofiles':('struct fileproc **',0,8,'-
>fileproc[]'),'fd_cdir':('struct vnode *',16,8,'->CWD'),'fd_lastfile':('int',36,4,'-
>fileproc[LAST_INDEX]')}
```

```
642                , 11:{'fd_ofiles':('struct fileproc **',0,8,'-
>fileproc[]'),'fd_cdir':('struct vnode *',16,8,'->CWD'),'fd_lastfile':('int',36,4,'-
>fileproc[LAST_INDEX]')}
643           }
644      }
645
646      def __init__(self, addr):
647           super(Filedesc, self).__init__(addr)
648
649      def getcwd(self):
650           cwd_ptr = unpacktype(self.smem, self.template['fd_cdir'], INT)
651           if self.validaddr(cwd_ptr):
652                return cwd_ptr
653           return None
654
655      def getfglobs(self):
656
657           # sometimes the fd is valid, but this array address is not (e.g. kernel_task)
658           ofiles_ptr = unpacktype(self.smem, Filedesc.template['fd_ofiles'], INT)
659           if ofiles_ptr == 0 or not(Struct.mem.is_valid_address(ofiles_ptr)):
660                return None
661
662           # construct a list of addresses from the fd_ofiles array
663           fd_lastfile    = unpacktype(self.smem, Filedesc.template['fd_lastfile'], INT)
664           ptr_size  = 4 if (Struct.arch == 32) else 8
665           fmt            = 'I' if (Struct.arch == 32) else 'Q'
666           fglobs         = {}
667           for i in xrange(fd_lastfile+1):
668
669                # **fd_ofiles is an array of pointers, read address at index i
670                fileproc_ptr = Struct.mem.read(ofiles_ptr+(i*ptr_size), ptr_size)
671                fileproc_addr = struct.unpack(fmt, fileproc_ptr)[0]
672
673                # not every index points to a valid file
674                if fileproc_addr == 0 or not(Struct.mem.is_valid_address(fileproc_addr)):
675                     continue
676
677                fileproc = Fileproc(fileproc_addr)
678                fileglob_ptr = fileproc.getfglob()
679
680                if fileglob_ptr != None:
681                     fglobs[i] = fileglob_ptr
682
683           return fglobs
684
685 # Vm_object --> Vnode_pager
686 class Vnode_pager(Struct):
687
688      TEMPLATES = {
689           32:{
690                10:{'vnode_handle':('struct vnode *',16,4,'->txt')}
691                , 11:{'vnode_handle':('struct vnode *',16,4,'->txt')}
```

```
692             },
693             64:{ # NOTE: 10.6/7x64 offset for vnode_pager edited manually 24 --> 32
694                  10:{'vnode_handle':('struct vnode *',32,8,'->txt')}
695                  , 11:{'vnode_handle':('struct vnode *',32,8,'->txt')}
696             }
697       }
698
699       def __init__(self, addr):
700             super(Vnode_pager, self).__init__(addr)
701
702       def gettxt(self):
703             txt_ptr = unpacktype(self.smem, self.template['vnode_handle'], INT)
704
705             # self may not actually be a vnode pager (there are other valid types), need to
706             # run several tests without generating warnings to be sure.
707             if txt_ptr == 0 or not(Struct.mem.is_valid_address(txt_ptr)):
708                  return None
709
710             # this pointer test ensures the target memory matches the vnode template
711             vnode = Vnode(txt_ptr)
712             if vnode.gettype() == -1 or vnode.getname() == None:
713                  return None
714
715             # return the pointer rather than vnode because duplicates will occur as a
    result
716             # of recursive calls in Vm_object
717             return txt_ptr
718
719 # Vm_map_entry --> Vm_object
720 class Vm_object(Struct):
721
722       TEMPLATES = {
723             32:{
724                  10:{'memq':('queue_head_t',0,8,'',{'next':('struct queue_entry
    *',4,4,'type test(vm_object)'),'prev':('struct queue_entry *',0,4,'type
    test(vm_object)')}),'shadow':('struct vm_object *',52,4,'-
    >vm_object(recurse)'),'pager':('memory_object_t',64,4,'->pager')}
725                  , 11:{'memq':('queue_head_t',0,8,'',{'next':('struct queue_entry
    *',4,4,'type test(vm_object)'),'prev':('struct queue_entry *',0,4,'type
    test(vm_object)')}),'shadow':('struct vm_object *',52,4,'-
    >vm_object(recurse)'),'pager':('memory_object_t',64,4,'->pager')}
726             },
727             64:{
728                  10:{'memq':('queue_head_t',0,16,'',{'next':('struct queue_entry
    *',8,8,'type test(vm_object)'),'prev':('struct queue_entry *',0,8,'type
    test(vm_object)')}),'shadow':('struct vm_object *',72,8,'-
    >vm_object(recurse)'),'pager':('memory_object_t',88,8,'->pager')}
729                  , 11:{'memq':('queue_head_t',0,16,'',{'next':('struct queue_entry
    *',8,8,'type test(vm_object)'),'prev':('struct queue_entry *',0,8,'type
    test(vm_object)')}),'shadow':('struct vm_object *',72,8,'-
    >vm_object(recurse)'),'pager':('memory_object_t',88,8,'->pager')}
730             }
```

```
731       }
732
733       def __init__(self, addr):
734            super(Vm_object, self).__init__(addr)
735            self.map = None
736
737            # this test determines wether self matches the struct vm_object template, or
      the
738            # vm_map template.
739            ptr1 = unpacktype(self.smem, self.template['memq'][4]['next'], INT)
740            ptr2 = unpacktype(self.smem, self.template['memq'][4]['prev'], INT)
741            if ptr1 == 0 or ptr2 == 0 \
742                 or not(Struct.mem.is_valid_address(ptr1)) \
743                 or not(Struct.mem.is_valid_address(ptr2)):
744
745                      # on failure, create map instance to be called recursively
746                      self.map = Vm_map(addr)
747
748       def gettxt(self):
749
750            # recurse on vm_map type
751            if self.map != None:
752                 return self.map.gettxt()
753
754            pager_ptr = unpacktype(self.smem, self.template['pager'], INT)
755
756            # objects for memory-mapped files keep the pager in the shadow object rather
757            # than the original, this test determines which self is.
758            if pager_ptr == 0 or not(Struct.mem.is_valid_address(pager_ptr)):
759
760                 shadow_ptr = unpacktype(self.smem, self.template['shadow'], INT)
761                 if shadow_ptr == 0 or not(Struct.mem.is_valid_address(shadow_ptr)):
762                      return [] # Vm_map expects an empty list, never None
763
764                 # make recursive call on shadow object
765                 shadow = Vm_object(shadow_ptr)
766                 return shadow.gettxt()
767
768            # the default case here wraps the return in a list for compatibility with the
769            # recursive map case.
770            pager = Vnode_pager(pager_ptr)
771            return [ pager.gettxt() ]      # NOTE: this may return [ None ] without error
772
773 # Vm_map_entry --> Vm_map_entry
774 # Vm_map       --> Vm_map_entry
775 class Vm_map_entry(Struct):
776
777       TEMPLATES = {
778            32:{
779                 10:{'links':('struct vm_map_links',0,24,'',{'prev':('struct vm_map_entry
      *',0,4,''),'next':('struct vm_map_entry *',4,4,'->vm_map_entry')}),'object':('union
      vm_map_object',24,4,'->vm_object')}
```

```
780                , 11:{'links':('struct vm_map_links',0,24,'',{'prev':('struct vm_map_entry
*',0,4,''),'next':('struct vm_map_entry *',4,4,'->vm_map_entry')}),'object':('union
vm_map_object',36,4,'->vm_object')}
781           },
782           64:{
783                10:{'links':('struct vm_map_links',0,32,'',{'prev':('struct vm_map_entry
*',0,8,''),'next':('struct vm_map_entry *',8,8,'->vm_map_entry')}),'object':('union
vm_map_object',32,8,'->vm_object')}
784                , 11:{'links':('struct vm_map_links',0,32,'',{'prev':('struct vm_map_entry
*',0,8,''),'next':('struct vm_map_entry *',8,8,'->vm_map_entry')}),'object':('union
vm_map_object',56,8,'->vm_object')}
785           }
786      }
787
788      def __init__(self, addr):
789           super(Vm_map_entry, self).__init__(addr)
790
791      def getnext(self):
792           return unpacktype(self.smem, self.template['links'][4]['next'], INT)
793
794      def gettxt(self):
795           vmobject_ptr = unpacktype(self.smem, self.template['object'], INT)
796
797           # some entries lack an object, check manually to prevent error
798           if vmobject_ptr == 0 or not(Struct.mem.is_valid_address(vmobject_ptr)):
799                return [] # Vm_map expects an empty list, never None
800
801           vm_object = Vm_object(vmobject_ptr)
802           return vm_object.gettxt()
803
804 # Vm_object --> Vm_map
805 # Task      --> Vm_map
806 class Vm_map(Struct):
807
808      TEMPLATES = {
809           32:{
810                10:{'hdr':('struct vm_map_header',12,32,'',{'links':('struct
vm_map_links',12,24,'',{'prev':('struct vm_map_entry *',12,4,''),'next':('struct
vm_map_entry *',16,4,'->vm_map_entry')}),'nentries':('int',36,4,'no. nodes')})}
811                , 11:{'hdr':('struct vm_map_header',12,44,'',{'links':('struct
vm_map_links',12,24,'',{'prev':('struct vm_map_entry *',12,4,''),'next':('struct
vm_map_entry *',16,4,'->vm_map_entry')}),'nentries':('int',36,4,'no. nodes')})}
812           },
813           64:{
814                10:{'hdr':('struct vm_map_header',16,40,'',{'links':('struct
vm_map_links',16,32,'',{'prev':('struct vm_map_entry *',16,8,''),'next':('struct
vm_map_entry *',24,8,'->vm_map_entry')}),'nentries':('int',48,4,'no. nodes')})}
815                , 11:{'hdr':('struct vm_map_header',16,56,'',{'links':('struct
vm_map_links',16,32,'',{'prev':('struct vm_map_entry *',16,8,''),'next':('struct
vm_map_entry *',24,8,'->vm_map_entry')}),'nentries':('int',48,4,'no. nodes')})}
816           }
817      }
```

```
818
819       def __init__(self, addr):
820             super(Vm_map, self).__init__(addr)
821
822       def gettxt(self):
823             vmmapentry_ptr = unpacktype(self.smem,
self.template['hdr'][4]['links'][4]['next'], INT)
824             nentries = unpacktype(self.smem, self.template['hdr'][4]['nentries'], INT)
825             ret_ptrs = []
826
827             # iterate over map entries in the linked-list and collect any backing vnode
ptrs
828             for i in xrange(nentries):
829
830                   if self.validaddr(vmmapentry_ptr):
831                         vm_map_entry = Vm_map_entry(vmmapentry_ptr)
832                         txt_ptrs = vm_map_entry.gettxt()
833
834                         for txt_ptr in txt_ptrs:
835
836                               # filter duplicates and check for null returns
837                               if txt_ptr != None and not(txt_ptr in ret_ptrs):
838                                     ret_ptrs.append(txt_ptr)
839
840                   vmmapentry_ptr = vm_map_entry.getnext()
841
842             # unique list of verified vnode pointers
843             return ret_ptrs
844
845 # Proc --> Task
846 class Task(Struct):
847
848       TEMPLATES = {
849             32:{
850                   10:{'map':('vm_map_t',24,4,'->vm_map')}
851                   , 11:{'map':('vm_map_t',20,4,'->vm_map')}
852             },
853             64:{
854                   10:{'map':('vm_map_t',40,8,'->vm_map'),}  # NOTE: 10.6x64 offset for vm_map
edited manually 36 --> 40
855                   , 11:{'map':('vm_map_t',32,8,'->vm_map')}# NOTE: 10.7x64 offset for vm_map
edited manually 28 --> 32
856             }
857       }
858
859       def __init__(self, addr):
860             super(Task, self).__init__(addr)
861
862       def gettxt(self):
863             vmmap_ptr = unpacktype(self.smem, self.template['map'], INT)
864
865             if self.validaddr(vmmap_ptr):
```

```
866                 vm_map = Vm_map(vmmap_ptr)
867                 return vm_map.gettxt()
868
869           return None
870
871 # Pgrp --> Session
872 class Session(Struct):
873
874      TEMPLATES = {
875           32:{
876                10:{'s_login':('char[]',28,255,'USER')}
877                , 11:{'s_login':('char[]',28,255,'USER')}
878           },
879           64:{
880                10:{'s_login':('char[]',48,255,'USER')}
881                , 11:{'s_login':('char[]',48,255,'USER')}
882           }
883      }
884
885      def __init__(self, addr):
886           super(Session, self).__init__(addr)
887
888      def getuser(self):
889           return unpacktype(self.smem, self.template['s_login'], STR).split('\x00',
     1)[0].strip('\x00')
890
891 # Proc --> Pgrp
892 class Pgrp(Struct):
893
894      TEMPLATES = {
895           32:{
896                10:{'pg_session':('struct session *',12,4,'->session')}
897                , 11:{'pg_session':('struct session *',12,4,'->session')}
898           },
899           64:{
900                10:{'pg_session':('struct session *',24,8,'->session')}
901                , 11:{'pg_session':('struct session *',24,8,'->session')}
902           }
903      }
904
905      def __init__(self, addr):
906           super(Pgrp, self).__init__(addr)
907
908      # skipped the full validator here because pg_session is the only pointer/target
909      def getuser(self):
910           session_ptr = unpacktype(self.smem, self.template['pg_session'], INT)
911
912           if self.validaddr(session_ptr):
913                session = Session(session_ptr)
914                return session.getuser()
915
916           return None
```

```
917
918 # _kernproc --> Proc
919 class Proc(Struct):
920
921      TEMPLATES = {
922           32:{
923                10:{'p_list':('LIST_ENTRY(proc)',0,8,'',{'le_next':('struct proc
    *',0,4,''),'le_prev':('struct proc **',4,4,'-
    >next')}),'p_pid':('pid_t',8,4,'PID'),'task':('void *',12,4,'->task'),'p_fd':('struct
    filedesc *',104,4,'->filedesc'),'p_textvp':('struct vnode *',388,4,'-
    >proc(exe)'),'p_comm':('char[]',420,17,'COMMAND'),'p_pgrp':('struct pgrp *',472,4,'-
    >pgrp')}
924                , 11:{'p_list':('LIST_ENTRY(proc)',0,8,'',{'le_next':('struct proc
    *',0,4,''),'le_prev':('struct proc **',4,4,'-
    >next')}),'p_pid':('pid_t',8,4,'PID'),'task':('void *',12,4,'->task'),'p_fd':('struct
    filedesc *',128,4,'->filedesc'),'p_textvp':('struct vnode *',412,4,'-
    >proc(exe)'),'p_comm':('char[]',444,17,'COMMAND'),'p_pgrp':('struct pgrp *',496,4,'-
    >pgrp')}
925           },
926           64:{
927                10:{'p_list':('LIST_ENTRY(proc)',0,16,'',{'le_next':('struct proc
    *',0,8,''),'le_prev':('struct proc **',8,8,'-
    >next')}),'p_pid':('pid_t',16,4,'PID'),'task':('void *',24,8,'->task'),'p_fd':('struct
    filedesc *',200,8,'->filedesc'),'p_textvp':('struct vnode *',664,8,'-
    >proc(exe)'),'p_comm':('char[]',700,17,'COMMAND'),'p_pgrp':('struct pgrp *',752,8,'-
    >pgrp')}
928                , 11:{'p_list':('LIST_ENTRY(proc)',0,16,'',{'le_next':('struct proc
    *',0,8,''),'le_prev':('struct proc **',8,8,'-
    >next')}),'p_pid':('pid_t',16,4,'PID'),'task':('void *',24,8,'->task'),'p_fd':('struct
    filedesc *',216,8,'->filedesc'),'p_textvp':('struct vnode *',680,8,'-
    >proc(exe)'),'p_comm':('char[]',716,17,'COMMAND'),'p_pgrp':('struct pgrp *',768,8,'-
    >pgrp')}
929           }
930      }
931
932      head = None
933
934      def __init__(self, addr):
935           super(Proc, self).__init__(addr)
936
937           if Proc.head == None:
938                Proc.head = addr
939
940           self.self_ptr       = addr    # store this for cycle detection by
    getfilelist()
941           self.filedesc_ptr   = None
942           self.exe_ptr        = None
943           self.pgrp_ptr       = None
944           self.pid            = -1
945
946      def next(self):
947           nxt_proc = unpacktype(self.smem, Proc.template['p_list'][4]['le_prev'], INT)
```

```
948
949             if nxt_proc == Proc.head:
950                  stderr.write("ERROR %s.%s encountered a circular list.\n"
%(self.__class__.__name__, sys._getframe().f_code.co_name))
951                  return None
952
953             elif nxt_proc != 0 and Struct.mem.is_valid_address(nxt_proc):
954                  return Proc(nxt_proc)
955
956             return None
957
958         # this method has evolved to check ALL requisite proc structure pointers
959         def valid(self):
960
961             # check *p_fd
962             filedesc_ptr = unpacktype(self.smem, self.template['p_fd'], INT)
963             if filedesc_ptr == 0 or not(Struct.mem.is_valid_address(filedesc_ptr)):
964                  return False
965
966             # check *p_textvp
967             exe_ptr = unpacktype(self.smem, self.template['p_textvp'], INT)
968             if exe_ptr == 0 or not(Struct.mem.is_valid_address(exe_ptr)):
969                  return False
970
971             # check *p_pgrp
972             pgrp_ptr = unpacktype(self.smem, self.template['p_pgrp'], INT)
973             if pgrp_ptr == 0 or not(Struct.mem.is_valid_address(pgrp_ptr)):
974                  return False
975
976             self.filedesc_ptr    = filedesc_ptr
977             self.exe_ptr         = exe_ptr
978             self.pgrp_ptr        = pgrp_ptr
979             return True
980
981         def setpid(self, pid):
982             self.pid = unpacktype(self.smem, Proc.template['p_pid'], INT)
983
984             while self.pid != pid:
985                  nxt_proc = unpacktype(self.smem, Proc.template['p_list'][4]['le_prev'],
INT)
986
987                  if nxt_proc == Proc.head:
988                       stderr.write("ERROR %s.%s encountered a circular list.\n"
%(self.__class__.__name__, sys._getframe().f_code.co_name))
989                       return False
990                  elif nxt_proc != 0 and Struct.mem.is_valid_address(nxt_proc):
991                       self.smem = Struct.mem.read(nxt_proc, Proc.ssize);
992                       self.pid = unpacktype(self.smem, Proc.template['p_pid'], INT)
993                  else:
994                       return False
995
996             filedesc_ptr = unpacktype(self.smem, self.template['p_fd'], INT)
```

```
997             if filedesc_ptr == 0:
998                  print "\nPID: %d (%s) has no open files." %(pid, self.getcmd())
999                  sys.exit()
1000            if not(Struct.mem.is_valid_address(filedesc_ptr)):
1001                 print "\nPID: %d (%s) has an invalid file descriptor." %(pid,
    self.getcmd())
1002                 sys.exit()
1003            if not self.valid():
1004                 print "\tPID: %d appears in the in process list, but is not compatible
    with lsof." %pid
1005                 sys.exit()
1006
1007            return True
1008
1009       def getfd(self):
1010            return self.filedesc_ptr
1011
1012       def getpid(self):
1013            if self.pid < 0:
1014                 return unpacktype(self.smem, Proc.template['p_pid'], INT)
1015            return self.pid
1016
1017       def getcmd(self):
1018            return unpacktype(self.smem, self.template['p_comm'], STR).split('\x00',
    1)[0].replace(' ', '\\x20').strip('\x00')
1019
1020       def getuser(self):
1021            pgrp = Pgrp(self.pgrp_ptr)
1022            return pgrp.getuser()
1023
1024       def gettxt(self):
1025            task_ptr = unpacktype(self.smem, self.template['task'], INT)
1026            task = Task(task_ptr)
1027            txt_ptrs = task.gettxt()
1028
1029            if not(self.exe_ptr in txt_ptrs):
1030                 txt_ptrs.append(self.exe_ptr)
1031
1032            return txt_ptrs
1033
1034  ################################### PRIVATE FUNCTIONS ####################################
1035
1036  # given a validated proc stucture, return a list of open files
1037  def getfilelistbyproc(proc):
1038
1039       filedesc   = Filedesc(proc.getfd())
1040       fglobs     = filedesc.getfglobs()
1041       filelist   = []
1042
1043       if fglobs == None:
1044            return []
1045
```

```
1046        cwd = Vnode(filedesc.getcwd())
1047        if cwd:
1048             filelist.append( (proc.getcmd(), proc.getpid(), proc.getuser(), "cwd  ",
1049                  cwd.gettype(), cwd.getdev(), cwd.getoff(-1), cwd.getnode(), cwd.getpath())
1050             )
1051
1052        txt_ptrs = proc.gettxt()
1053        for txt_ptr in txt_ptrs:
1054             txt = Vnode(txt_ptr)
1055             filelist.append( (proc.getcmd(), proc.getpid(), proc.getuser(), "txt  ",
1056                  txt.gettype(), txt.getdev(), txt.getoff(-1), txt.getnode(), txt.getpath())
1057             )
1058
1059        # iterate over fileglob structures, note: items() is unsorted by default
1060        for fd, fglob in sorted(fglobs.items()):
1061
1062             # this has been observed as an invalid pointer even when fileproc is not
1063             if fglob == 0 or not Struct.mem.is_valid_address(fglob):
1064                  continue
1065
1066             fileglob = Fileglob(fglob)
1067
1068             # full support for VNODE (1) only, otherwise, just print ftype for verbose
1069             ftype = fileglob.gettype()
1070
1071             # exclude file if type cannot be resolved
1072             if ftype == -1:
1073                  continue
1074
1075             if ftype != 'VNODE':
1076                  if Struct.verb:
1077                       filelist.append( (proc.getcmd(), proc.getpid(), proc.getuser(),
1078                            fileglob.getmode(fd), ftype, -1, -1, -1, -1)
1079                       )
1080                  continue
1081
1082             vnode_ptr = fileglob.getdata()
1083             if vnode_ptr == None:
1084                  continue
1085
1086             vnode = Vnode(vnode_ptr)
1087             filelist.append( (proc.getcmd(), proc.getpid(), proc.getuser(),
1088                  fileglob.getmode(fd), vnode.gettype(), vnode.getdev(),
1089                  vnode.getoff(fileglob.getoff()), vnode.getnode(), vnode.getpath())
1090             )
1091
1092        return filelist
1093
1094   ################################### PUBLIC FUNCTIONS ###################################
1095
1096   # build list of processes with open files, and return the aggregate listing
1097   def getfilelist(mem, arch, kvers, proc_head, pid, vflag):
```

```
1098
1099      Struct.mem           = mem
1100      Struct.arch          = arch
1101      Struct.kvers    = kvers
1102      Struct.verb          = bool(vflag)
1103
1104      proc = Proc(proc_head)
1105      if pid > -1:
1106           if proc.setpid(pid): # returns True on success
1107                return getfilelistbyproc(proc)
1108           print "\tPID: %d not found in process list." %pid
1109           sys.exit()
1110
1111      ptr_list = []   # this list catches cycles in the linked list (known to occur)
1112      proclist = []
1113      while proc:
1114
1115           if proc.self_ptr in ptr_list:  # test for cycle
1116                stderr.write("ERROR getfilelist(): proc linked-list cycles, results may be
     incomplete.\n")
1117                break
1118           ptr_list.append(proc.self_ptr)
1119
1120           if proc.valid():
1121                proclist.append(proc)
1122           proc = proc.next()
1123
1124      fullfilelisting = []
1125      for proc in proclist:
1126           fullfilelisting += getfilelistbyproc(proc)
1127
1128      return fullfilelisting
1129
1130 # given the output of getfilelist(), build a string matrix as input to columnprint()
1131 def printfilelist(filelist):
1132      headerlist = ["COMMAND", "PID", "USER", "  FD  ", "TYPE", "DEVICE", "SIZE/OFF",
     "NODE", "NAME"]
1133      contentlist = []
1134
1135      for file in filelist:
1136           line = ["%s" %file[0]]
1137           line.append("%d" %file[1])
1138           line.append("%s" %file[2])
1139           line.append("%s" %file[3])
1140           line.append("%s" %file[4])
1141           line.append("%s" %file[5])
1142           line.append("%s" %file[6])
1143           line.append("%d" %file[7])
1144           line.append("%s" %file[8])
1145           contentlist.append(line)
1146
1147      #columnprint(headerlist, contentlist)
```

```
1148
1149        # use optional max size list here to match default lsof output, otherwise specify
1150        # lsof +c 0 on the command line to print full name of commands
1151        mszlist = [9, -1, -1, -1, -1, -1, -1, -1, -1]
1152        columnprint(headerlist, contentlist, mszlist)
```

Bibliography

Apple Inc. (2009, February 4). Mac OS X ABI Mach-O file format reference. *Mac OS X Developer Library*. Retrieved from https://developer.apple.com/library/mac/#documentation/developertools/conceptual/MachORuntime/Reference/reference.html

Apple Inc. (2011, March 21). lsof(8) Mac OS X manual page. *Mac OS X Developer Library*. Retrieved from http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man8/lsof.8.html

Architecture Technology Corporation. (2011). Mac Memory Reader. Retrieved from http://cybermarshal.com/index.php/cyber-marshal-utilities/mac-memory-reader

Becher, M., Dornseif, M., & Klein, C. N. (2005, May). FireWire: All your memory are belong to us [slides]. *CanSecWest*. Retrieved from https://cansecwest.com/core05/2005-firewire-cansecwest.pdf

Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux kernel* (3rd ed.). Sebastopol, CA: O'Reilly Media.

Burdach, M. (2004, March 22). Forensic analysis of a live Linux system, Pt. 1. *SecurityFocus*. Retrieved from http://www.securityfocus.com/infocus/1769

Burdach, M. (2006, January). Finding digital evidence in physical memory [slides]. *Black Hat Federal*. Retrieved from https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Burdach/bh-fed-06-burdach-up.pdf

Carvey, H. (2009). *Windows forensic analysis DVD toolkit* (2nd ed.). Burlington, MA: Syngress.

Case, A. (2011a, March 1). Bringing Linux support to Volatility. *Corporate Blog: Digital Forensics Solutions*. Retrieved from http://dfsforensics.blogspot.com/2011/03/bringing-linux-support-to-volatility.html

Case, A. (2011b, July). Linux memory analysis with Volatility [slides]. *Open Memory Forensics Workshop*. Retrieved from http://digitalforensicssolutions.com/papers/omfw.pdf

Case, A. (2011c, August). Investigating live CDs using Volatility and physical memory analysis [slides]. *Black Hat USA*. Retrieved from https://media.blackhat.com/bh-us-11/Case/BH_US_11_Case_Linux_Slides.pdf

Case, A., Cristina, A., Marziale, L., Richard, G. G., & Roussev, V. (2008). FACE: Automated digital evidence discovery and correlation. *Digital Investigation*, 5(Supplement), S65-S75.

Case, A., Marziale, L., & Richard, G. G. (2010). Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7(Supplemental), S32-S40.

Case, A., Marziale, L., Neckar, C., & Richard, G. G. (2010). Treasure and tragedy in kmem_cache mining for live forensics investigation. *Digital Investigation*, 7(Supplement), S41-S47.

Casey, E., Fellows, G., Geiger, M., & Stellatos, G. (2011). The growing impact of full disk encryption on digital forensics. *Digital Investigation*, 8(2), 129-134.

Choi, J., Savoldi, A., Gubian, P., Lee, S., & Lee, S. (2008, April). Live forensic analysis of a compromised Linux system Using LECT (Linux Evidence Collection Tool). *International Conference on Information Security and Assurance*. Busan, Korea: IEEE Computer Society.

Digital Forensics Research Workshop, Inc. (2008). DFRWS 2008 Forensics Challenge Overview. *Inside DFRWS 2008*. Retrieved from http://www.dfrws.org/2008/challenge/index.shtml

Dolan-Gavitt, B. (2008a). Forensic analysis of the Windows registry in memory. *Digital Investigation*, 5(Supplement), S26-S32.

Dolan-Gavitt, B. (2008b, April 16). Finding kernel global variables in Windows. *Personal Blog*. Retrieved from http://moyix.blogspot.com/2008/04/finding-kernel-global-variables-in.html

Donnini, G. (2012, April 3). Operating system market share, March 2011 update. Chitika *Insights*. Retrieved from http://insights.chitika.com/2012/operating-system-market-share-march-2012-update/

Garrison, T. (2011, September 18). Mac OS Lion forensic memory acquisition using IEEE 1394. *Personal Blog*. Retrieved from http://www.frameloss.org/2011/09/18/firewire-attacks-against-mac-os-lion-filevault-2-encryption/

Gladyshev, P., & Almansoori, A. (2010, October). Reliable acquisition of RAM dumps from Intel-based Apple Mac computers over FireWire. *Second International ICST Conference*. Abu Dhabi, United Arab Emirates: Springer.

Gladyshev, P., & Almansoori, A. (2011). Goldfish. *UCD Centre for Cybersecurity and Cybercrime Investigation*. Retrieved from http://cci.ucd.ie/goldfish

Gorman, M. (2004). *Understanding the Linux virtual memory manager*. Upper Saddle River, NJ: Prentice Hall.

Graham, R. D. (2011, February 24). Thunderbolt: Introducing a new way to hack Macs. *Errata Security*. Retrieved from http://erratasec.blogspot.com/2011/02/thunderbolt-introducing-new-way-to-hack.html

Haruyama, T., & Suziki, H. (2012, March). One-byte Modifications for Breaking Memory Forensic Analysis. *Black Hat Europe*. Retrieved from https://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf

Hay, B., & Nance, K. (2009). Live analysis: Process and challenges. *IEEE Security & Privacy*, 7(2), 30-37.

Hermann, U. (2008, August 14). Physical memory attacks via Firewire/DMA. *Personal Blog*. Retrieved from http://www.hermann-uwe.de/blog/physical-memory-attacks-via-firewire-dma-part-1-overview-and-mitigation

Inman, K., & Rudin, N. (2002). The origin of evidence. *Forensic Science International*, 126(1), 11-6. Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/11955825

Inoue, H., Adelstein, F., & Joyce, R. A. (2011). Visualization in testing a volatile memory forensic tool. *Digital Investi*gation, 8(Supplement), S42-S51.

Intel Corporation. (2012). Intel 64 and IA-32 Architectures Software Developer's Manual. Retrieved from http://download.intel.com/products/processor/manual/325462.pdf

Kessler, T. (2012, February 1). FileVault 2 easily decrypted, warns Passware. *CNET Reviews*. Retrieved http://reviews.cnet.com/8301-13727_7-57369983-263/filevault-2-easily-decrypted-warns-passware/

Kollár, I. (2010). *Forensic RAM dump image analyser* (Master's thesis). Charles University, Prague, Czech Republic.

Kubasiak, R. R., & Morrissey, S. (2009). *Mac OS X, iPod, and iPhone forensic analysis DVD toolkit*. Burlington, MA: Syngress.

Law, F. Y. W., Chow, K. P., Kwan, M. Y. K., & Lai, P. K. Y. (2007, December). Consistency issue on live systems forensics. *International Conference on Future Generation Communication and Networking*. Jeju, Korea: IEEE Computer Society.

Leat, C. (2011). *Forensic analysis of Mac OS X memory* (Master's thesis). University of Westminster, London, England.

Lee, K. (2012). volafox: Memory analyzer for Mac OS X & BSD. Retrieved from https://code.google.com/p/volafox/

Leyden, J. (2003, October 17). Caffrey acquittal a setback for cybercrime prosecutions. *The Registe*r. Retrieved from http://www.theregister.co.uk/2003/10/17/caffrey_ acquittal_a_setback/

Ligh, M. H., Adair, S., Hartstein, B., & Richard, M. (2011). *Malware analyst's cookbook and DVD: Tools and techniques for fighting malicious code*. Indianapolis, IN: Wiley.

Maartmann-Moe, C. (2012, February 6). Adventures with daisy in Thunderbolt-DMA-land: Hacking Macs through the Thunderbolt interface. *Break & Enter*. Retrieved from http://www. breaknenter.org/2012/02/adventures-with-daisy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface/

Makinen, J. (2008, February 29). Automated OS X Macintosh password retrieval via FireWire. *Personal Blog*. Retrieved from http://www.juhonkoti.net/2008/02/29/ automated-os-x-macintosh-password-retrieval-via-firewire

Malard, A. (2011, August 3). H@CKMacOSX: Tips and tricks for Mac OS X hack. *Personal Blog*. Retrieved from http://sud0man.blogspot.com/2011/08/hackmacosx-tips-and-tricks-for-mac-os-x.html

Mandia, K., Prosise, C., & Pepe, M. (2003). *Incident response and computer forensics* (2nd ed.). Emeryville, CA: McGraw-Hill.

McGrew, W. (2008, March 28). msramdmp: McGrew Security RAM dumper. *McGrew Security*. Retrieved from http://www.mcgrewsecurity.com/tools/msramdmp/

Miller, C., & Zovi, D. D. (2009). *The Mac hacker's handbook*. Indianapolis, IN: Wiley.

Movall, P., Nelson, W., & Wetzstein, S. (2005, April). Linux physical memory analysis. *USENIX Annual Technical Conference*. Retrieved from http://static.usenix.org/ event/usenix05/tech/freenix/full_papers/movall/movall.pdf

National Institute of Justice. (2008, April). *Electronic crime scene investigation: A guide for first responders* (2nd ed.). Washington, DC: Mukasey, M. B., Sedgwick, J. L., & Hagy, D. W.

National Institute of Standards and Technology. (2008, March). *Computer security incident handling guide: Recommendations of the National Institute of Standards and Technology*. (NIST Special Publication 800-61, Revision 1). Gaithersburg, MD: Scarfone, K., Grance, T., & Masone, K.

National Institute of Standards and Technology. (2006, August). *Guide to integrating forensic techniques into incident response* (NIST Special Publication 800-86). Gaithersburg, MD: Kent, K., Chevalier, S., Grance, T., & Dang, H.

Net Applications. (2012, March). Operating system market share: March 2012. *NetMarketShare.com*. Retrieved from http://www.netmarketshare.com/operating-system-market-share.aspx

Petroni, N. L., Walters, Aa., Fraser, T., & Arbaugh, W. (2006). FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4), 197-210.

Russinovich, M. (2011, May 18). Handle v3.46. *Windows Sysinternals*. Retrieved from http://technet.microsoft.com/en-us/sysinternals/bb896655

SANS Institute. (2008). *Covering the tracks on Mac OS X Leopard*. Retrieved from https://www.sans.org/reading_room/whitepapers/apple/covering-tracks-mac-os-leopard_32993

SANS Institute. (2012). *Forensics and Incident Response Summit Agenda*. Retrieved from http://www.sans.org/forensics-incident-response-summit-2012/ agenda.pdf

Schonfeld, E. (2012, January 11). U.S. PC shipments slip 6 percent in Q4, while Apple's jump 21 percent. *TechCrunch*. Retrieved from http://techcrunch.com/2012/01/11/ gartner-pc-shipments-slip-6-percent-q4-apple-jump-21-percent/

Schuster, A. (2011, June 7). Mac OS X memory analysis with volafox. *Personal Blog*. Retrieved from http://computer.forensikblog.de/en/2011/06/mac_os_x_memory_ analysis_with_volafox.html

Sesek, R. (2012, January 25). Debugging Mach Ports. *Personal Blog*. Retrieved from http://robert.sesek.com/thoughts/2012/1/debugging_mach_ports.html

Singh, A. (2006a). Accessing kernel memory on the x86 version of Mac OS X. In *Mac OS X internals: A systems approach* (bonus content). Retrieved from http://osxbook.com/book/bonus/chapter8/kma

Singh, A. (2006b). *Mac OS X internals: A systems approach*. Upper Saddle River, NJ: Addison-Wesley.

Siracusa, J. (2011, July 20). Mac OS X 10.7 Lion: the Ars Technica review. *Ars Technica*. Retrieved from http://arstechnica.com/apple/reviews/2011/07/mac-os-x-10-7.ars/18

Solomon, J., Huebner, E., Bem, D., & Szezynska, M. (2007). User data persistence in physical memory. *Digital Investigation*, 4(2), 68-72.

Suiche, M. (2010, February). Mac OS X physical memory analysis. *Black Hat DC*. Retrieved from https://www.blackhat.com/presentations/bh-dc-10/Suiche_Matthieu/Blackhat-DC-2010-Advanced-Mac-OS-X-Physical-Memory-Analysis-wp.pdf

Urrea, J. M. (2006). *An analysis of Linux RAM forensics* (Master's thesis). Retrieved from Defense Technical Information Center. (Accession No. ADA445300)

Valenzuela, I. (2011, January 28). Mac OS forensics how-to: Simple RAM acquisition and analysis with Mac Memory Reader. *SANS Computer Forensics Blog*. Retrieved from http://computer-forensics.sans.org/blog/2011/01/28/mac-os-forensics-howto-simple-ram-acquisition-analysis-mac-memory-reader-part-1

Vidas, T. (2011, January). MemCorp: An open data corpus for memory analysis. *44th Hawaii International Conference on System Sciences*. Koloa, Hawai: IEEE Computer Society.

Volatile Systems, LLC (2012). Volatility: An advanced memory forensics framework. Retrieved from https://code.google.com/p/volatility/

Webb, T. (2010, August 7). Creating a OS X live IR CD-ROM. *Personal Blog*. Retrieved from https://irhowto.wordpress.com/2010/08/07/creating-a-os-x-live-ir-cd-rom

Witherden, F. (2010, October 15). Memory forensics over the IEEE 1394 interface. *ibforensic1394 Project*. Retrieved from https://freddie.witherden.org/tools/libforensic1394

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
14 Jun 2012	Master's Thesis	28 Jun 2010 – 14 Jun 2012

4. TITLE AND SUBTITLE

Forensic Memory Analysis for Apple OS X

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hay, Andrew F.

5d. PROJECT NUMBER
JON# 11G258

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCO/ENG/12-17

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
ATTN: Joseph A. Carozzoni, Research Scientist
AFRL/RIGD Cyber Science Branch Bldg 3 Suite J2 A-21
525 Brooks Road, Rome, NY 13441-4505
315-330-7796, joseph.carozzoni@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RIGD

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Analysis of raw memory dumps has become a critical capability in digital forensics because it gives insight into the state of a system that cannot be fully represented through traditional disk analysis. Interest in memory forensics has grown steadily in recent years, with a focus on the Microsoft Windows operating systems. However, similar capabilities for Linux and Apple OS X have lagged by comparison. The volafox open source project has begun work on structured memory analysis for OS X. The tool currently supports a limited set of kernel structures to parse hardware information, system build number, process listing, loaded kernel modules, syscall table, and socket connections. This research addresses one memory analysis deficiency on OS X by introducing a new volafox module for parsing file handles. When open files are mapped to a process, an examiner can learn which resources the process is accessing on disk. This listing is useful for determining what information may have been the target for exfilitration or modification on a compromised system. Comparing output of the developed module and the UNIX lsof (list open files) command on two version of OS X and two kernel architectures validates the methodology used to extract file handle information.

15. SUBJECT TERMS
digital forensics; memory analysis; apple os x; volafox; file handles

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Peterson, Gilbert L., Ph.D
U	U	U	UU	166	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-6565, ext 4281 (gilbert.peterson@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18