Shrinking the Ocean: Formalizing I/O Methods in Modern Operating Systems

Matthew B. Gerber, Research Scientist, the Institute for Simulation and Training
John J. Leeson, Ph.D, Associate Professor of Computer Science
University of Central Florida

Abstract

Currently, it is not practical for any single software system to perform forensically acceptable verification of the contents of all possible file systems on a disk, let alone the contents of more esoteric peripherals. Recent court decisions that require judges to restrict testimony based on their understanding of the validity of the science behind it will only make such verification even more difficult. This problem, critical to forensic examiners, is actually symptomatic of a larger problem, which lies partly in the domain of digital forensics and partly in the domain of pure computer science. Lack of verifiability, along with a host of other problems, points to inadequate formal description of file systems and I/O methodology. A review of the literature finds, in fact, that little effort has been put into such formalization. We assert that a constructive formalization of peripheral input and output for a computer can address this and several other concerns.

1. Introduction

Currently, it is not practical for any single software system to perform verification of the contents of all possible file systems on an IDE, SCSI, USB, or IEEE 1394 hard disk drive—let alone the contents of more esoteric peripherals.

Examiners are not without means of navigating this sea of information—forensic scientists use various innovative tools and jury-rig various effective techniques to enable them to obtain vital data, and newer versions of these tools can read many of the most common file system formats. (NTI, Guidance, U.S. Treasury)

Verifying that each and every one of these tools and (especially) techniques are forensically acceptable, however, is not an easy task in the laboratory, let alone the courtroom. Recent court decisions that both empower and require judges to restrict testimony based on their understanding of the validity of the science behind it will only make such verification even more difficult. (*Daubert*, *Kumho*)

In the long run, we believe it is not sufficient merely to obtain data. Examiners must begin to obtain data in a *forensically acceptable* manner. We propose that this means using techniques that have survived the accepted processes of scientific peer review in credible publications, and that should therefore be acceptable with little difficulty under the *Daubert* standard for scientific evidence.

This problem is most obviously critical to forensic examiners. However, it is actually symptomatic of a larger problem, which lies partly in the domain of digital forensics and partly in the domain of pure computer science. To shed light on this problem, we first recall how computers handle information in general.

In the strictest sense, a *computer* is only the microprocessor and physical (or *primary*) memory that reside on the motherboard of what is typically known as a PC. Computers need to receive input from, and generate output to, a variety of *peripherals*. Examples of peripherals are fixed disk drives, removable magnetic disks, removable optical disks, network interface cards, printers, analog modems, ISDN modems, video cards, sound input devices, sound output devices, keyboards, pointing devices; the list goes on. All of these peripherals store, transmit or receive data, but before anything can be done with that data, it has to wind up in primary memory so the processor can look it at.

All of the above peripherals are also part of a hierarchy that extends both above and below them. To create an example that we will use throughout this paper, a typical input request would be to retrieve a range of bytes from a file on a fixed disk. To do this, a typical PC-architecture computer must interface with and/or understand, in turn:

- The PCI expansion bus.
- The IDE device controller residing on the PCI expansion bus.
- The file system of the partition on the fixed disk drive that contains the file.

At this point, the computer may retrieve the necessary metadata from the file system to find the bytes it is looking for, and retrieve those bytes into primary memory.

The elements of this hierarchy are disparate and scattered. For example, a computer running a modern operating system, such as Microsoft Windows or Linux, will typically have separate device drivers for the PCI expansion bus, the IDE device controller, and the file system that contains the file. These drivers will be unrelated, interacting code: each is dependent on each of the others, and an error in any of them will cause the others to fail in unexpected—and often nearly untraceable—ways. Were we discussing navigation, it would be as though different types of maps and entirely different reference points were required to deal with the conditions of current and weather around every individual island in the sea.

The elements of this hierarchy are also highly duplicated. A recent version of the Linux operating system contained file system drivers for over twenty file systems, several of which were over 200K in source size—yet the functionality of every file system is fundamentally similar. Various IDE controllers must have their own drivers as well. If the fixed disk drive resides on a SCSI bus instead of an IDE bus, then the hierarchy of drivers is different yet again, for an essentially similar function. (Torvalds)

As more and different peripherals (such as fixed disks and their controllers) and means of accessing those peripherals (such as file systems) become available, the difficulties faced in accessing them all will multiply; the ocean will only grow wider and deeper, and require even more maps. This will become a concern to anyone who wishes to create or maintain stable, reliable software systems capable of interfacing with most available peripherals; even today, it is of concern to two significant groups.

- Law enforcement officers must be able not only to read information from every conceivable type of peripheral, but to do so in a verifiable, duplicable, and completely non-destructive manner. Current systems are simply inadequate to this task: most serious attempts to produce verifiable results involve rebooting a computer into MS-DOS and running highly expensive tools that work with very few types of peripherals.
- Maintainers of current operating systems find I/O to be increasingly troublesome: Microsoft has gone on record blaming up to 80 percent of crashes in some versions of Windows on device drivers. Since most Windows device drivers are actually written by third parties, it has become increasingly difficult for Microsoft to quality-control their own operating system with respect to its I/O subsystems; driver certification programs have lessened this problem, but do not solve it in a fundamental sense.
- Designers of new operating systems find I/O to be an even worse problem. Peripherals that work on the most popular operating systems are expected to work on all, and regardless of where the necessary effort lies, the operating system will be blamed if a peripheral fails to work with it. As the number of available peripherals explodes, and as peripheral vendors focus their driver design efforts more and more tightly, it becomes more and more difficult for any operating systems that do not already have drivers to obtain them.

We contend that all of these concerns are symptomatic of the same problem. Lack of verifiability, scattershot design techniques, high amounts of duplication of effort, and low reusability all point to inadequate description. A review of the literature finds, in fact, that little effort has been put into formalization of this particular domain.

2 Literature Review

2.1 ISO 7498-1

2.1.1 Review

We begin by reviewing the accepted standard in a domain that *has* been well-formalized. ISO 7498-1 is the International Standards Organization's reference model of networking: the Open Systems Interconnection model. The document spends its first five sections describing in some detail the assumptions and thought processes that led to the model, which is finally described in the sixth and seventh sections. It is these sections that primarily interest us.

The Open Systems Interconnection Environment (OSIE) is defined in seven layers of operation, numbered highest at the most abstract and lowest at the hardware level. They are discussed in the reverse order of their numbering. The first two, the most abstract, are the least defined by the document.

- Layer 7 is the **application layer.** It provides services for end-user applications to access the OSIE, and is the only layer that does. No layer of the OSIE sits above the application layer, and the application layer is the only entry point to the OSIE.
- Layer 6 is the **presentation layer.** The presentation layer defines the syntax of information being transferred, translating it between the application layer and the rest of the OSIE.

The middle three layers are the heart of the document's definition of networking. These are the layers that provide the services typically associated with a network by network software developers and users.

- Layer 5 is the **session layer.** This is the layer that has the responsibility for opening, maintaining, and closing connections (*sockets* in TCP/IP).
- Layer 4 is the **transport layer.** This is the layer that handles error detection and correction, sequencing control and reordering, and flow control.
- Layer 3 is the **network layer.** This is the layer that handles network routing, relaying, and gateways between sub-networks.

The bottom two layers, like the top two, are also less thoroughly defined.

- Layer 2 is the **data link layer.** This is the layer that handles data transmission between individual points on the network. It may also handle routing within a sub-network.
- Layer 1 is the **physical layer.** This is the layer that describes the physical communications media.

The layers are very rigidly defined; to the point that, for example, as far as connection-based and connectionless services are defined, it is restrictively specified which layers may convert between the two. The document defines explicitly the services provided by each layer to the layer above it, and the services used by each layer from the layer below it.

2.1.2 Commentary

The OSI networking model's domain is obviously different from the one we consider here: it concerns one computer communicating with another, while we are interested in a computer communicating with components of itself.

However, it is difficult to overstate the emphasis that the OSI model has had on its domain, the networking community; it has been called a defining moment in the development of networking as a science. Its formalization was the catalyst that allowed computer networks—largely, until then, based on proprietary and incompatible technologies—to reliably interoperate. (Shumaker, 2002)

2.2 Technical Standards for Drive Controllers

As we have intimated, there is a lack of formal description of interface mechanisms; however, there is no such shortage of technical documentation. The two most common interface systems for fixed disk drives are well-documented.

- The AT Attachment with Packet Interface (ATA/ATAPI) standard is the technical name of the hard drive interface system commonly known as IDE (for Intelligent Drive Electronics). ATA/ATAPI is maintained by Technical Committee T13 of the International Committee on Information Technology Standards (INCITS); its finalized documents are available from the American National Standards Institute (ANSI) and its drafts may be downloaded via the Web. (INCITS-T13)

- The Small Computer Systems Interface (SCSI) standard is maintained by Technical Committee T10 of the INCITS, which handles lower-level interfaces. As with T13, T10's draft documents are available online and its finished products may be purchased from ANSI. (INCITS-T13)

2.3 Papers

Some effort has been made toward formalizing file systems, but it has largely been in terms of very specific file systems or aspects of those file systems. As far as we have determined, no generalized formalization of file systems exists.

Ciancarini, Fogli and Gaspari describe *Gammalog*, a declarative language for problem solving that includes the concepts of coordination. They illustrate its expressive power by including a simple "operating system" written in the language. Actually calling it an operating system, however, is overstating the case; the functions that actually read and write files are not described in the paper. (Ciancarini et al., 2000)

Heisel describes an attempt to specify the user view of the UNIX file system in the Z modeling language. The attempt is interesting to us because of the formal level at which it models the file system in question. It is, however, concerned with what the user sees, not what is actually stored within the file system itself. Actual discussion of the fine points of the model would require a specification of the Z language, which Heisel does not give and which we will not give here. (Heisel, 1995)

Heydon and Tygar describe *Miró*, a formal system for specifying and checking security constraints under UNIX. Miró is not theoretical; its designers intend it to be implemented and used by system administrators. Its domain is limited to security. (Heydon & Tygar, 1994)

Miró defines two languages: an *instance language* to create security *configurations*, which are simply matrices of subjects versus objects on a file system with each cell being "grant" or "deny"; and a *constraint language* for security *policies*, whereby each policy is a set of constraints which is in turn a set of configurations, and a configuration is consistent with a policy if it is within each of that policy's configurations.

The instance language is a (relatively) simple set of named boxes and lines. A box can specify a set of users or a set of files, and can contain other boxes that are subsets. A "user" box can have a directed line drawn from it to a "file" box, the line specifying either the granting or the denying of certain accesses. More specific arrows have higher priority. Each box has a type that gives it certain attributes; the types are specified in an object-oriented manner, children inheriting attributes from their parents.

A constraint specifies a pattern of instance pictures, just as a language specifies a pattern of strings. A *box pattern* specifies, in a predicate language, the characteristics of the boxes it matches. *Semantics arrows* specify access permissions to be matched between boxes that match the box patterns they connect. *Containment arrows* specify containment relationships to be matched between boxes that match the box patterns they connect.

The remainder of the paper describes the implementation of the software system itself, and is beyond the scope of this review. Miró is interesting to us for its formal specification of a methodology used by file systems, but in the end its precepts and mechanisms are limited to security.

3. The Solution: A New Formalization

As said in the first section, we assert that a constructive formalization of peripheral input and output for a computer will, if subscribed to, address all of the concerns that we have raised here. Starting in this paper, and continuing in a series of papers that will follow, we will design this constructive formalization. We begin by considering what we need to formalize.

3.1 On the Organization of File Systems

A file system can be seen as a tree of directories with sequential files as its leaves. All modern file systems—including FAT, FAT32, NTFS, HFS, HFS+, ext2, ext3, reiserfs, xfs, and others far too numerous to list—operate in this fashion.

All of these systems also have a date and time stamp for when the file was last modified. Some also have one for when it was initially created; some even have one for when it was last accessed.

Other forms of metadata are more fragmented. File permissions do not exist *per se* in FAT or HFS, but flags that indicate whether the file should be "read-only" do. ext2 uses the UNIX model of file system permissions, whereas NTFS implements access control lists. HFS and HFS+ permit files to be explicitly typed by a four-byte code, whereas the other file systems rely on the file's "extension"—usually the last few characters of its name, following a period—to know its type. NTFS also permits files to contain alternate data streams, allowing a file to be its own pseudo-directory of sorts.

In summary, there are many differences between file systems, but far more commonalities. All of these file systems (and we have listed all of the major ones in use on desktop computers) store essentially the same information: hierarchical directory structures, file names, timestamp information, permission-style metadata, and the sequentially ordered bytes comprising the data of each file. Indeed, an examination of the source code for the Linux operating system finds that the interface from the kernel to each file system driver is identical: code to handle individual file systems is abstracted into well-defined modules.

We propose to take this abstraction a step further.

3.2 Hadley

We intend to create a formal, declarative language intended for the purpose of describing input and output systems. The language will be sufficiently robust to be usable to prove properties of I/O systems, and will be sufficiently descriptive to be usable for direct implementation of I/O systems described in it. The language is named Hadley in honor of John Hadley, the British inventor of the sextant, one of the earliest devices used for seagoing navigation.

Hadley version 1.0 will describe file systems, and be implemented as a universal file system driver for the BSD UNIX operating system. Future versions of Hadley will handle broader swathes of the I/O chain (such as IDE, SCSI, IEEE 1394, and USB controllers) and support more operating systems.

3.3 The Probable Limitations of Hadley

The chief limitation of any high level model is that it cannot efficiently model the primitives it uses to execute itself. As an example, any engine designed to execute Hadley would have to itself execute in primary memory, and (to work with various expansion buses) have access to the computer's clock primitives.

Through sufficiently tortured logic it might be possible to treat memory, the clock, or even both as provided by I/O systems, but it will likely simplify matters deeply to not do so. Given that simplification, and the fact that it does not seem unreasonable to ask for memory space and clock time as primitives, we will treat them as such.

This means that while the full version of Hadley, once complete, could describe any peripheral or—if implemented—serve as the I/O mechanism for any operating system, it likely could not adequately describe memory management or the clock signal.

3.4 The Advantages of Hadley

Hadley 1.0, as a constructive model capable of generally describing file systems, will by itself produce many advantages. We describe a few here.

- Given the ability to prove that a file system is described by its description in Hadley, properties proven about that description would necessarily hold true for the file system itself.
- Given a fully constructive model, a representation of it would, as we have inferred, be sufficient for a properly designed program to execute a Hadley description itself—i.e., read from and write to a file system based solely on its description.
- Any operating system that properly supported Hadley could support any file system for which a description was written, without the need for recompilation of the operating system or installation of new and possibly suspect binary modules.
- It could be relatively easily proven (not merely demonstrated) that an execution of a given description accessed a given file system accurately and without modifying its contents, hence satisfying law enforcement's need for verifiable ability to access file systems.
- Any improvement in the efficiency of the operating system module that interprets Hadley would be immediately felt by all file systems. Recently, the Linux operating system added geometry-based optimization for write operations to the ext3 file system; given support for Hadley, such an addition could be made to every supported file system at once. (Tweedie, 1998)

Once Hadley has moved beyond file systems, the same advantages expand accordingly to all forms of input and output that Hadley covers. It is at this stage that Hadley could, for example, replace the drivers for individual IDE and SCSI controllers that exist in most modern operating systems.

Extending further, it is entirely conceivable that a single, relatively small binary code module in an operating system could suffice to handle all forms of space and stream based input and output, and that all improvements made to that module would be felt by all such forms of input and output. This state would be an obvious improvement, for both formal and practical purposes, over the current situation of I/O subsystems.

4. References

New Technologies Inc. *Safeback* (software package). Information available at URL: http://www.forensics-intl.com/safeback.html.

Guidance Software. *Encase* (software package). Information available at URL: http://www.guidancesoftware.com/.

United States Department of the Treasury. *iLook* (software package). Information available at URL: http://ilook-forensics.org/.

Daubert v. Merrill Dow Pharmaceuticals (509 U.S. 579, 1993)

Kumho Tire Company v. Patrick Carmichael (526 U.S., 1999)

Linus Torvalds et al. *Linux* (operating system kernel). Available at URL: http://www.kernel.org/.

International Organization for Standardization. "Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model". Publication ISO/IEC 7498-1:1994.

Dr. Randall Shumaker. Interview (2002).

International Committee on Information Technology Standards Technical Committee T13: AT Attachment. Information available at URL: http://www.t13.org.

International Committee on Information Technology Standards Technical Committee T10: Lower-Level Interfaces. Information available at URL: http://www.t10.org.

Paolo Ciancarini, Daniela Fogli, and Mauro Gaspari. "A declarative coordination language". *Computer Languages* 26 (2-4), pp. 125-163 (2000).

Maritta Heisel. "Specification of the Unix file system: A comparative case study". *Algebraic Methodology and Technology Lecture Notes in Computer Science* 936, pp. 475-488 (1995).

Allan Heydon and J. D. Tygar. "Specifying and Checking Unix Security Constraints". *Computing Systems* 7 (1), pp. 91-112 (1994).

Stephen C. Tweedie. "Journaling the Linux ext2fs Filesystem". *LinuxExpo* 1998. Available at URL: ftp://ftp.uk.linux.org:/pub/linux/sct/fs/jfs/journal-design.ps.gz.

5. About the Authors

Matthew B. Gerber is a research scientist at the Institute for Simulation and Training at the University of Central Florida. He has a Master's degree in Computer Science from UCF and is currently pursuing a Ph.D. there. His research interests include computer forensics, formal systems and software interoperability.

Dr. John J. Leeson is an Associate Professor of Computer Science at the University of Central Florida and Assistant Director for Digital Evidence of the National Center for Forensic Science. He has a Ph.D. in Mathematics from the University of Miami. His research interests include computer forensics and systems software. Dr. Leeson has received Computer Forensic training from the National White Collar Crime Center and the International Association of Computer Investigative Specialists (IACIS). He holds CFCE certification from IACIS.