Leveraging Forensic Tools for Virtual Machine Introspection

Brendan Dolan-Gavitt Bryan Payne Wenke Lee
School of Computer Science
Georgia Institute of Technology
{brendan,bdpayne,wenke}@cc.gatech.edu

Abstract

Virtual machine introspection (VMI) has formed the basis of a number of novel approaches to security in recent years. Although the isolation provided by a virtualized environment provides improved security, software that makes use of VMI must overcome the *semantic gap*, reconstructing high-level state information from low-level data sources such as physical memory. The digital forensics community has likewise grappled with semantic gap problems in the field of forensic memory analysis (FMA), which seeks to extract forensically relevant information from dumps of physical memory. In this paper, we will show that work done by the forensic community is directly applicable to the VMI problem, and that by providing an interface between the two worlds, the difficulty of developing new virtualization security solutions can be significantly reduced.

1 Introduction

Commodity operating systems (OSes) are currently extremely difficult to secure. Weak OS-level protections have made it difficult to ensure that security tools running on the host can effectively perform their functions without interference from malware running on the system. Recently, virtualization has been proposed as a solution to this problem: by moving security software into an isolated environment provided by a trusted hypervisor, we can ensure that code running in the guest operating system cannot tamper with it.

However, this isolation comes at a price. Although the security software is now free from interference, it is likewise prevented from accessing the APIs available to applications running inside the untrusted OS. Critical information required to make security decisions is available only through *virtual machine introspection (VMI)*: the security tool must reconstruct high-level semantic knowledge about the guest operating system based on low-level sources such as physical memory and CPU registers. This divide, known as the *semantic gap*, makes the development of virtualized security solutions more difficult than their in-guest counterparts.

Similar problems are faced in the field of *forensic memory analysis (FMA)*, which seeks to extract forensic information from dumps of physical memory. This technique is an attractive alternative to traditional live response, as it offers similar information with higher integrity and repeatability. Commercial products [5, 6, 10] and open-source tools [13, 17] have emerged that offer the ability to analyze dumps of memory from Windows and Linux systems.

In this paper, we examine the semantic gap problem from the perspective of FMA. By creating simple software that provides access to guest memory in a format forensic tools can understand, we can immediately benefit from their analysis, allowing the development of VMI tools to proceed much more rapidly. We will show several benefits of this combination by using forensic tools to detect malware and rootkits on a live virtual machine running Windows XP. Aside from the interface between tools, this can be accomplished with no new software—a significant reduction in the effort needed to provide security in a virtualized environment.

We will also demonstrate that building on existing forensic software can allow researchers to rapidly create new VMI tools. By making use of existing capabilities in a popular open-source FMA framework, we were able to reimplement VMWall [14], a virtualized security tool that can securely filter traffic based on the originating application. Because the code for listing open connections and active processes already existed in the forensic software, our implementation is an order of magnitude smaller (in SLOC). This demonstrates the marked reduction in effort that can be achieved by applying existing forensic software to VMI problems.

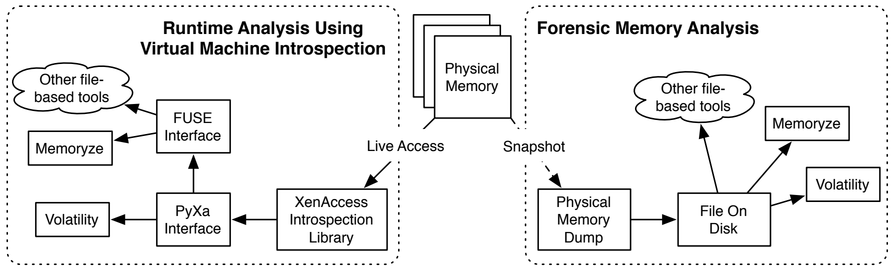


Figure 1: Both virtual machine introspection (VMI) and forensic memory analysis (FMA) start with the same view of memory. One of the main differences is that VMI operates at runtime whereas FMA is static. Even so, the same tools can be used to address the semantic gap problem in both settings.

2 Related Work

Virtual machine introspection has been a critical part of many recent virtualization-based approaches to security. First introduced by Garfinkel and Rosenblum [4], introspection allows security software to gain an understanding of the current state of the guest virtual machine. APIs such as XenAccess [11] and VMSafe [16] have appeared that can be used to provide the low-level access required to read the virtual machine state, and researchers have created applications ranging from intrusion detection systems [4] and firewalls [14] to malware analysis frameworks [1,7] and zero-day attack analysis platforms [9].

On the forensic side, work has focused on finding ways to bridge the semantic gap and recover security-relevant artifacts from physical memory. These efforts have resulted in tools that are able to find processes and threads [13], detect DLL injection [18], recover files mapped in memory [15], and extract information about the Windows registry [2]. Although this work does not solve the semantic gap problem in general, it provides valuable insight into the internals of widely deployed operating systems that can provide critical data for security tools.

Our own work unites these two fields, showing how the work done by the forensic community can be directly applied to the problems of virtual machine introspection. By demonstrating several interfaces between forensic tools and virtual machines, we hope to ease the difficulty of introspection and spur the creation of new tools.

3 Comparing FMA and VMI

VMI software runs in an isolated virtualized environment and monitors the state of other VMs. This isolation protects it from tampering by software inside the monitored VM, making it an attractive way to implement security software. VMI-based monitoring is performed online and focuses on detecting security events as they occur. FMA, by contrast, typically takes place after a security incident is suspected to have occurred. An investigator acquires an image of physical memory and then performs offline analysis, extracting information about the system state to explain the incident. In this section, we will examine the key similarities (shown in Figure 1) between these two fields that allow approaches developed for FMA to be used for VMI.

In order to reconstruct information about the state of the OS, both approaches rely primarily on physical memory as a source of data. However, modern operating systems operate using virtual memory; thus, to appropriately interpret OS-level data structures, VMI and FMA applications must be able to translate virtual addresses to their physical location in memory. On x86-based systems, a set of *page tables* are used to perform the virtual-to-physical mapping. Once the page tables for the kernel address space are located, virtual memory becomes accessible and more meaningful data about the state of the system can be extracted.

Both VMI and FMA applications must deal with semantic gap issues. Because native OS APIs are not available, information about the state of the system must be gathered by locating and examining the internal data structures that the in-guest APIs use. This is currently a difficult and manual process, particularly on closed source operating systems such as Windows, where details of data structures must be obtained through reverse engineering. Because both virtual machine introspection and forensic memory analysis examine physical memory, advances made by the forensic community in understanding these structures can be directly applied to virtualization security.

The advances made by the forensic community in reconstructing high-level state from physical memory dumps are not entirely general, however. Most work has focused on recovering information about specific operating systems and versions; in particular, many FMA tools currently only support Windows XP with Service Pack 2. Although support for other operating systems is increasing (some commercial tools support multiple versions of Windows, and several open-source FMA frameworks now support Linux), FMA does not provide a complete solution to the semantic gap problem. In the absence of a general solution, however, the information provided by FMA can enable a variety of new VMI applications and allow researchers to avoid duplicating effort when implementing new systems.

Because VMI examines a *live* system, it also has access to information beyond what is available in a forensic context. In addition to the static view of physical memory provided by a memory dump, VMI applications can watch the state of the system as it changes over time. This allows the use of techniques that are not possible in offline analysis: for example, Antfarm [8] tracks the value of the CR3 register (which holds the address of the page directory for the current active process) to determine what processes are running inside the virtual machine. Since a memory image gives a view of the system state at a single point in time, this technique will only work in a live environment.

Although much of the same information is available with both VMI and FMA, there are some difficulties associated with virtual machine introspection that are not present with offline forensic analysis. Because the system is still running, the CPU and memory state will change as analysis is performed unless the CPU is suspended before analysis (which can negatively impact performance). By contrast, FMA tools need only be concerned with memory volatility at the time the snapshot is taken: although the system continues to run throughout the acquisition process, the results are static and can be examined offline.

Despite these differences, forensic memory analysis tools can be of great value to virtual machine introspection. As we will show, work done in the forensic community to bridge the semantic gap can be directly applied to the problems faced by introspection (though the converse is not necessarily true). By applying forensic tools and techniques in a virtualized setting, we can reduce the time needed to develop new virtualization security applications.

4 Forensic VMI Interfaces

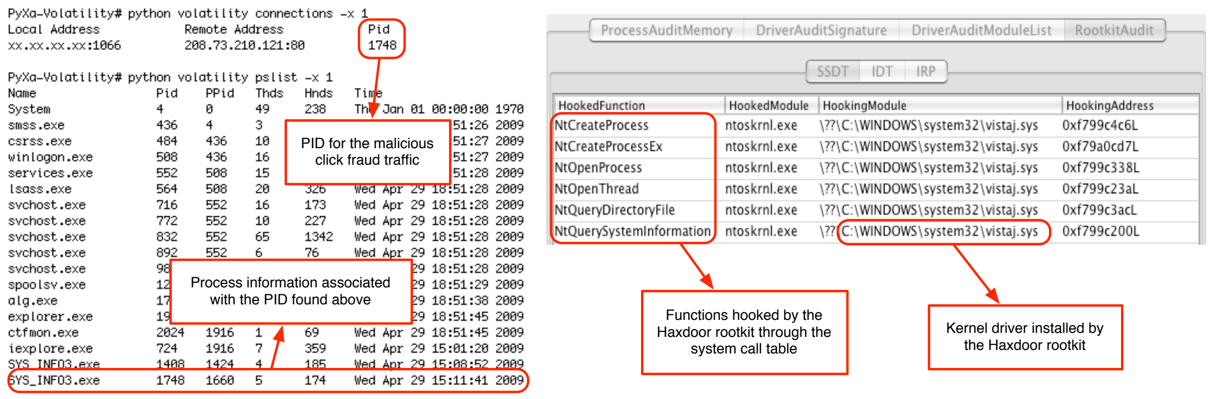
To make use of forensic tools for VMI, the forensic software needs to access the memory of the guest virtual machine (VM). There are two ways to accomplish this: modifying existing forensic software to extend its capabilities and presenting the memory of a guest VM in a format that FMA tools can understand. The former would allow FMA software to take advantage of the live nature of VMI; however, the latter requires no modifications to existing tools.

We have implemented two general methods for allowing existing applications to interface with the memory of a guest virtual machine as a *filesystem interface* and an *extension API*. The filesystem interface exposes guest memory as a regular file on the host, allowing FMA tools that process memory dumps to operate on guest memory with no modifications. The extension API requires a small number of changes to the forensic software to enable access to guest memory using an introspection library such as XenAccess [11]. Although the filesystem interface is more general, the extension API can provide a forensic tool with additional information that may improve its analysis capabilities. Both interfaces have been freely released as part of XenAccess so they can benefit other researchers.

Both interfaces are built on top of XenAccess, which does the actual work of reading the guest physical memory. We built a Python C extension around this library called PyXa, which presents a Xen VM as a Python object with a `read` method that reads a single page of physical memory from the guest, and a helper method `get_cr3` that retrieves the current value of the `CR3` register from the guest's virtual CPU. We created this wrapper in Python to allow prototype VMI applications to be built more quickly and because the most popular open-source memory analysis tool, Volatility [17], is written in Python. Volatility can enumerate active processes, open network connections, loaded kernel modules, and other data that are relevant to both VMI and FMA.

The filesystem interface was implemented using FUSE [3], a library that allows filesystems to be built without writing kernel code. Because FUSE filesystems can be written in high-level languages such as Python, we were able to write a filesystem that uses PyXa to expose guest memory as a regular file in just 156 SLOC. When an application attempts to read from the file, FUSE calls the `read` method of our pseudo-file, which in turn calls PyXa's `read` method to read the requested data from the guest's physical memory. This allows any application to treat virtual machine memory as though it were a regular memory image on disk.

Because the filesystem interface presents the guest's memory as a file, no changes to forensic applications are required. However, this simple interface does not provide any extra information from VMI that might be useful to FMA tools. To allow forensic applications to benefit from additional knowledge such as the state of CPU registers, we extended an existing open-source memory analysis framework, Volatility, to access virtual machine memory by interfacing directly with PyXa. This extension to Volatility ($\approx$ 80 SLOC) allows the forensic framework to access the `CR3` register and begin performing virtual address translation immediately; normally, a scan of physical memory is required to locate the initial page directory and bootstrap the address translation process.



```
PyXa-Volatility# python volatility connections -x 1
Local Address           Remote Address         Pid
xx.xx.xx.xx:1066        208.73.210.121:80      1748

PyXa-Volatility# python volatility pslist -x 1
Name                Pid    PPid   Thds   Hnds   Time
System              4      0      49     238    Thu Jan 01 00:00:00 1970
smss.exe            436    4      3                         51:26 2009
csrss.exe           484    436    10                        51:27 2009
winlogon.exe        508    436    16                        51:27 2009
services.exe        552    508    15                        51:28 2009
lsass.exe           564    508    20     326    Wed Apr 29 18:51:28 2009
svchost.exe         716    552    16     173    Wed Apr 29 18:51:28 2009
svchost.exe         772    552    10     227    Wed Apr 29 18:51:28 2009
svchost.exe         832    552    65     1342   Wed Apr 29 18:51:28 2009
svchost.exe         892    552    6      76     Wed Apr 29 18:51:28 2009
svchost.exe         98                                 29 18:51:28 2009
spoolsv.exe         12                                 29 18:51:29 2009
alg.exe             17                                 29 18:51:38 2009
explorer.exe        19                                 29 18:51:45 2009
ctfmon.exe          2024   1916   1      69     Wed Apr 29 18:51:45 2009
iexplore.exe        724    1916   7      359    Wed Apr 29 15:01:20 2009
SYS_INFO3.exe       1408   1424   4      185    Wed Apr 29 15:08:52 2009
SYS_INFO3.exe       1748   1660   5      174    Wed Apr 29 15:11:41 2009
```

PID for the malicious click fraud traffic
Process information associated with the PID found above
ProcessAuditMemory
DriverAuditSignature
DriverAuditModuleList
RootkitAudit
SSDT
IDT
IRP
Functions hooked by the Haxdoor rootkit through the system call table
Kernel driver installed by the Haxdoor rootkit

HookedFunction	HookedModule	HookingModule	HookingAddress
NtCreateProcess	ntoskrnl.exe	\??\C:\WINDOWS\system32\vistaj.sys	0xf799c4c6L
NtCreateProcessEx	ntoskrnl.exe	\??\C:\WINDOWS\system32\vistaj.sys	0xf79a0cd7L
NtOpenProcess	ntoskrnl.exe	\??\C:\WINDOWS\system32\vistaj.sys	0xf799c338L
NtOpenThread	ntoskrnl.exe	\??\C:\WINDOWS\system32\vistaj.sys	0xf799c23aL
NtQueryDirectoryFile	ntoskrnl.exe	\??\C:\WINDOWS\system32\vistaj.sys	0xf799c3acL
NtQuerySystemInformation	ntoskrnl.exe	\??\C:\WINDOWS\system32\vistaj.sys	0xf799c200L

Figure 2: (Left) Detecting which process is responsible for click fraud network traffic using XenAccess for virtual machine introspection and Volatility to interpret the raw memory data. (Right) Rootkit detection on a running host using Memoryze connected to a virtual machine introspection backed file.

These two interfaces can provide a rich, high-level view into the state of the guest virtual machine. As we will see in the next section, these simple interfaces allow many introspection-based security applications to be trivially realized with off-the-shelf forensic software. Process and threads can be enumerated using PTFinder, open files and active network connections can be examined using Volatility, and commercial memory analysis software such as Memoryze [10] and HBGary Responder [6] can be used to detect rootkit activity in the guest VM.

5 Application Examples

Bridging the semantic gap opens the door to many new applications. Even though the forensic memory analysis applications and libraries discussed above do not provide the same rich application programmer interfaces (APIs) that are provided by the operating system, they do provide enough information to enable the rapid development of useful security applications. In this section, we show two ideas that can be rapidly built using PyXa and the FUSE module. However, these ideas are simply examples to illustrate the utility of this approach. The information available through forensic memory analysis tools enables the exploration of a wide variety of security applications.

5.1 Identifying the Source of Malicious Network Traffic

Using PyXa to connect Volatility to XenAccess, we are able to extract a wide variety of information from a *running* Windows XP VM. This information includes any open connections, a list of the running processes, DLLs loaded for each process, open files for each process, kernel modules, open sockets, thread objects, and more. Using the list of open connections, combined with the list of running processes, we can identify which process inside the VM is responsible for particular network traffic leaving the VM.

For this scenario, we assume that some existing mechanism can be used to detect malicious or unusual network traffic leaving the VM. For the purposes of our testing, we generated this malicious traffic by installing bots inside the VM. The bots we used included click fraud bots, spam bots, and trojans (`DR/Click.Delf.BW`, `AdClicker-AD`, `DR/Click.HSP.A.2`, `AdClicker-BY`, `Spammer:Win32/Cutwail.gen!B`, `Trojan.Inject.IA`, and `Trojan:Win32/Waledac.gen!A`). In each case we first identified the malicious network traffic using a network packet analyzer. Next, using Volatility on the running VM, we identified the open connection associated with this network traffic and the process responsible for the open connection. In each case, we were able to identify the process responsible for the malicious traffic. Figure 2 shows the output produced while running `DR/Click.Delf.BW`.

Volatility is designed to be a tool operated from the command line. However, with some simple modifications, it can also be used as a Python library. As a result, the type of investigation described above can be automated, resulting in a network firewall that allows or denies traffic based on the process that produced the traffic. This functionality is essentially the same as VMWall [14]; however, by leveraging the semantic information provided by Volatility this application can be written much more quickly.

To demonstrate this point, we wrote a Python script that uses the user-space packet filtering feature of `iptables` to make decisions about whether to allow or deny packets based on the originating application. When a packet is sent from the guest VM, its source and destination IP and ports are extracted, and Volatility is used to determine the originating application. An application whitelist is then used to decide if the packet is allowed onto the external network. By making use of existing code in Volatility, this prototype took only 68 SLOC in Python and about an hour of labor to implement. By contrast, the source code to VMWall (written in C) consists of 1094 SLOC.

5.2 Rootkit Detection

Many FMA tools are not able to integrate with XenAccess using PyXa. This includes FMA tools that are not written in Python, that are not open source, or that access memory through another abstraction. One abstraction commonly used with FMA tools is to work from a single file containing a physical memory dump. As described in Section 4, our filesystem interface provides this abstraction, allowing the use of many FMA tools to operate on a live system using VMI. One such tool is Memoryze, a freeware product by Mandiant [10].

Memoryze finds rootkits by identifying hidden drivers and hooks in the system call table, interrupt descriptor tables (IDT) and driver function tables (IRP). To demonstrate this capability working on a running system using our FUSE adapter running with XenAccess, we installed several rootkits inside the VM (`Trojan:Win32/DriverBypass`, `Backdoor:Win32/Haxdoor`, `Trojan.Dropper.Farfli.G`). After installing each rootkit, we ran the `HookDetection.bat` script provided by Memoryze, using our FUSE adapter as the input file. For each rootkit that we tested, we detected system call table hooking. The `Trojan:Win32/DriverBypass` rootkit also used IRP hooking. Figure 2 shows the output produced while running `Backdoor:Win32/Haxdoor`.

6 Discussion

The two applications described above are intended solely to demonstrate the possibilities of integrating FMA tools with VMI. We do not view the specific applications as being novel. Instead, we emphasize that these applications were built very rapidly by leveraging the semantic knowledge contained within the FMA tools. Given the ease with which they were built, it is notable that each test case resulted in detecting and acquiring additional information about the malware. Indeed, we view this as an indication that the overarching idea of malware detection on a running machine using VMI is a powerful and promising direction warranting further research.

The semantic gap continues to pose significant problems for researchers working with VMI. Now that the foundations have been laid for using VMI in real world applications with both passive [4, 11] and active [1, 12] monitoring, researchers are working on applications that utilize this technology. However, for each application researchers are spending a significant amount of time addressing the semantic gap problem. The result is often a solution involving many static, hard-coded values that allow prototype code to work on a narrow set of applications or operating systems. Using the techniques that we describe in this paper, researchers can more easily focus on the creation of their applications without being encumbered by the semantic gap issue.

We readily acknowledge that the FMA tools do not represent a *general* solution to the semantic gap problem. There is still a significant amount of research to be done to resolve the semantic gap problem. Existing solutions, such as the FMA tools, provide a relatively small amount of information when compared with the complete set of operating system and application level programmer's interfaces typically available in a modern system. There is also a need to provide coverage across a wider set of software versions, perhaps even automating the memory analysis process to allow support for new software versions. In addition, future security applications will need to monitor application memory as well as operating system memory. There are still a variety of open, hard problems to solve before we will have a general solution to the semantic gap problem.

Until there is a general solution to the semantic gap problem, the FMA tools are the best option available to security researchers. In many cases, the information provided by these tools is sufficient to enable the rapid creation of research prototypes. Using this technique, researchers can continue to explore applications that use VMI while the orthogonal semantic gap problem is being addressed. In cases where the information provided by existing FMA tools is insufficient to support a security application, one can still choose to extend an existing FMA tool. For example, writing a module for Volatility allows one to build on top of the semantic information already provided in Volatility. Either way, using FMA tools to assist with VMI saves time and limits the amount of redundant work.

7 Conclusion

A significant amount of recent security research has focused on using virtual machine introspection to create security applications that can operate with some degree of isolation from malware. These applications must overcome the semantic gap problem by interpreting the target system's memory to understand high-level details about the system's operation. This semantic gap problem remains a difficult, open research problem today. However, working and publishing separately from the traditional security conferences, the digital forensics community has put significant effort into developing systems to extract information from physical memory snapshots. In this paper, we demonstrated that these forensic systems can be used as a partial solution to the semantic gap problem when accessing running systems through virtual machine introspection. As a result, security researchers using these techniques can more rapidly develop prototype software and test their research ideas.

References

[1] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.

[2] B. Dolan-Gavitt. Forensic Analysis of the Windows Registry in Memory. In *Proceedings of the Digital Forensic Research Workshop (DFRWS)*, 2008.

[3] FUSE. FUSE: Filesystem in userspace. `http://fuse.sourceforge.net/`.

[4] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[5] G. M. Garner. KnTTools. `http://gmgsystemsinc.com/knttools/`.

[6] HBGary. Responder Pro: Physical Memory and Malware Analysis Application. `https://www.hbgary.com/products-services/responder-professional/`.

[7] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[9] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2005.

[10] MANDIANT. Memoryze. `http://www.mandiant.com/software/memoryze.htm`.

[11] B. D. Payne, M. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of the Annual Computer Security Applications Conference*, 2007.

[12] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[13] A. Schuster. Searching for processes and threads in Microsoft Windows memory dumps. In *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS)*, 2006.

[14] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2008.

[15] R. van Baar, W. Alinka, and A. van Ballegooija. Forensic memory analysis: Files mapped in memory. *Proceedings of the Digital Forensic Research Workshop (DFRWS)*, 2008.

[16] VMWare, Inc. VMWare VMSafe security technology. `http://www.vmware.com/technology/security/vmsafe.html`.

[17] A. Walters. The Volatility framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.

[18] A. Walters. FATKit: Detecting Malicious Library Injection and Upping the "Anti". Technical report, 4TΦ Research Laboratories, July 2006.