SANDIA REPORT

SAND2012-7818
Unlimited Release
Printed September 2012

Simplifying Virtual Machine Introspection Using LibVMI

Bryan D. Payne

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

WIZ, Inc. EXHIBIT - 1039
WIZ, Inc. v. Orca Security LTD.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-7818
Unlimited Release
Printed September 2012

Simplifying Virtual Machine Introspection Using LibVMI

Bryan D. Payne
Information Systems Analysis Center
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-1248

Abstract

Ensuring the security of a computer system requires the careful integration of many components. Key among these is security monitoring. Recent research trends show an increasing acceptance of external host-based monitoring techniques such as virtual machine introspection (VMI), a technique for viewing the runtime state of a virtual machine (VM). VMI's primary drawbacks include performance and the semantic gap problem (i.e., understanding the low-level information available through VMI). This report describes work performed under an Early Career Laboratory Directed Research and Development (LDRD) project that aimed to address these two key challenges. Our results are promising, with significant performance improvements and a much more usable VMI programming environment. This work resulted in the creation and release of LibVMI, an open source software project based on the author's previous work with the XenAccess library.

ACKNOWLEDGMENTS

This work was made possible through the Early Career LDRD program at Sandia National Labs. This program has proven to be an excellent tool for lab recruitment and for integrating new technical staff into the lab.

I would also like to thank Tan Thai for serving as a mentor for this project, and Matthew Leinhos for helping with the software development of LibVMI.

Finally, I'd like to thank the Cyber Security Early Career LDRD PI group for guidance and support on a variety of levels throughout this program.

CONTENTS

1. Introduction 7
2. LibVMI 9
   2.1 KVM Support 10
   2.2 API Improvements 11
   2.3 Performance 12
   2.4 64-bit Guest Support 14
3. PyVMI and Volatility Integration 15
   3.1 PyVMI: A Python Wrapper for LibVMI 15
   3.2 PyVMI Address Space for Volatility 15
4. Future Work 17
5. Conclusions 18
A1. Distribution 19

FIGURES

Figure 1 LibVMI enables the creation of a single VMI application that runs in multiple virtualization contexts. LibVMI is extensible to support any virtualization platform, but currently supports Xen, KVM, and physical memory snapshots. 9
Figure 2 LibVMI currently works with Xen, KVM, and physical memory snapshots. 9
Figure 3 High-level view of the LibVMI software stack. The portions in blue represent code written for this project. 10
Figure 4 Sampling of the current LibVMI API. The complete API contains a variety of additional convenience functions designed to make VMI development easier. 11
Figure 5 LibVMI's page cache algorithm balances memory overhead with performance, ensuring that the related data structures never grow too large. 13
Figure 6 LibVMI performance before our cache optimizations. 13
Figure 7 LibVMI performance after our cache optimizations. 14
Figure 8 While the details in this picture are too small to read, the key point is that significant work happens for each API call. In this case, the vmi_read_ksym call must handle reads around page boundaries, resolve the kernel symbol, translate the kernel symbol to a physical address, and perform the actual read from the VMM. 14
Figure 9 Software stack with PyVMI wrapper on top of the C language LibVMI library. The portions in blue represent code written for this project. 15
Figure 10 Software stack with Volatility address space plugin. The portions in blue represent code written for this project. 16

NOMENCLATURE

API	application programming interface
LDRD	laboratory directed research and development
KVM	kernel-based virtual machine (see www.linux-kvm.org)
PI	principal investigator
VM	virtual machine
VMI	virtual machine introspection
VMM	virtual machine monitor, analogous to a hypervisor
Xen	open source hypervisor from Univ of Cambridge (see www.xen.org)

1. INTRODUCTION

Previous virtual machine introspection (VMI) research has focused on the underlying mechanics (e.g, accessing memory pages) or extracting higher-level semantics from software (e.g., memory analysis). The work performed on this LDRD addressed the practical problems associated with VMI application development by bridging these two previous research areas. We approached the problem through the creation of LibVMI, a virtual machine introspection library based on the related XenAccess library. In addition, we provided integration between LibVMI and Volatility, a forensic memory analysis framework, to drastically simplify the creation of VMI applications.

LibVMI provides a useful application programming interface (API) for reading to and writing from a virtual machine's memory. It also provides a variety of utility functions that are useful to VMI developers. All of this functionality works for VMs running under either of the two most popular open source virtualization platforms: Xen and KVM. LibVMI programs can also use a static memory snapshot as a data source. This flexibility allows developers to create VMI applications once and have them work in each of these settings without modification. We discuss LibVMI in Section 2.

Volatility is an open source memory analysis framework. It is popular in the forensic memory analysis community where the goal is to understand the information within a single, static memory snapshot. Volatility can easily be extended to acquire its memory data from a source other than a file through a mechanism called address space plugins. We wrote an address space plugin for Volatility that enabled using LibVMI for memory access. Since Volatility is written in Python, this required also writing a Python wrapper for the LibVMI API. With this functionality in place, one can easily write new VMI applications using Volatility. We discuss the Volatility – LibVMI integration in Section 3.

This LDRD ended earlier than scheduled because the principal investigator (PI) decided to leave Sandia National Labs to pursue another job. This left some work unfinished. In Section 4, we will talk about this unfinished work as potential future work.

Finally, in Section 5 we provide some conclusions on this LDRD project.

2. LIBVMI

LibVMI provides a useful application programming interface (API) for reading to and writing from a virtual machine's memory. It also provides a variety of utility functions that are useful to VMI developers. All of this functionality works for VMs running under either of the two most popular open source virtualization platforms: Xen and KVM. LibVMI programs can also use a static memory snapshot as a data source. This flexibility allows developers to create VMI applications once and have them work in each of these settings without modification, as shown in Figures 1 and 2.

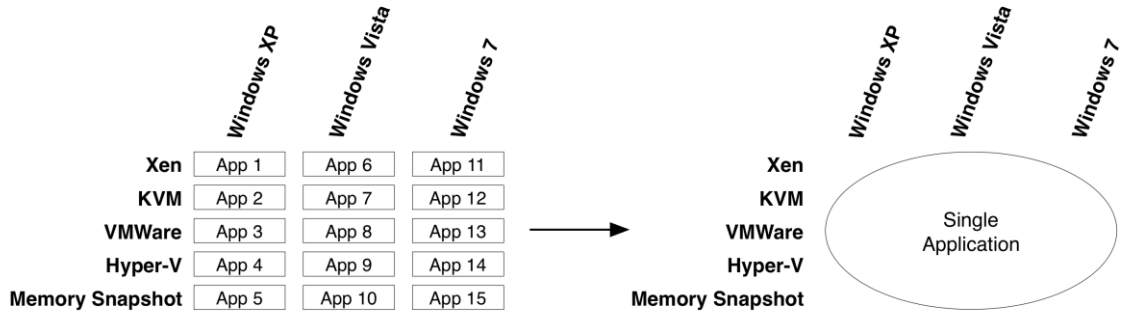

Figure 1 LibVMI enables the creation of a single VMI application that runs in multiple virtualization contexts. LibVMI is extensible to support any virtualization platform, but currently supports Xen, KVM, and physical memory snapshots.

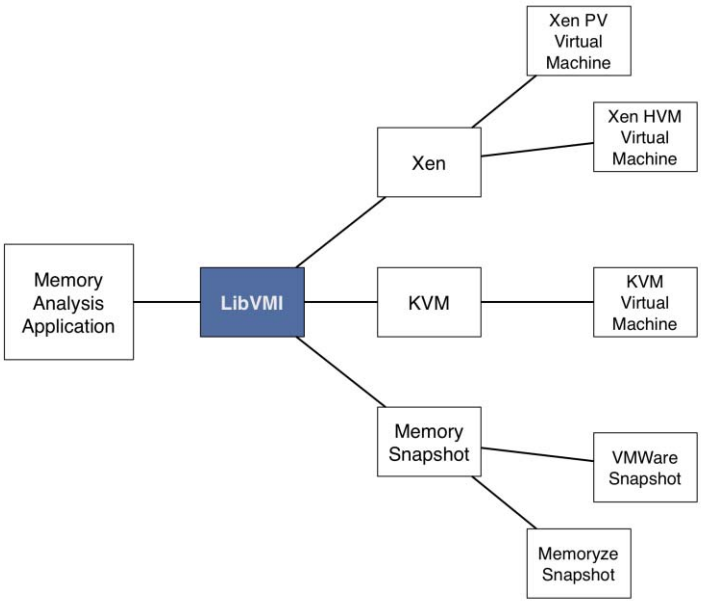

Figure 2 LibVMI currently works with Xen, KVM, and physical memory snapshots.

LibVMI evolved from the XenAccess project. XenAccess provided lower-level VMI capabilities for VMs running in Xen. With XenAccess, only 32-bit VM operating systems were supported. Furthermore, access to memory required the VMI developer to use XenAccess to manually map guest VM pages, operate on the pages, and then unmap the pages. The last

XenAccess release was version 0.5. LibVMI used this release as a starting point. Note that the PI for this LDRD, Bryan D. Payne, is also the creator of XenAccess.

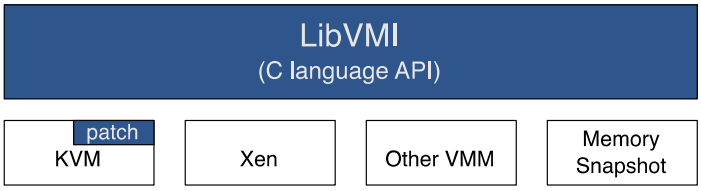

Figure 3 High-level view of the LibVMI software stack. The portions in blue represent code written for this project.

The key areas of improvement for LibVMI under the LDRD program include:

- Refactoring the code to support KVM, and to make supporting other virtualization platforms very simple.
- Improving the API to greatly simplify VMI development. Specifically, replace manual memory mapping with read and write functions that behave as expected to a POSIX developer.
- Improving the overall performance of the library.
- Adding support for 64-bit VM guest operating systems.
- Adding the pyvmi wrapper library (discussed in Section 3)
- Improving Volatility integration (discussed in Section 3)
- Fixing a variety of bugs ranging from correctness to memory leaks.

We discuss the first four bullet points in more detail below.

2.1 KVM Support

While Xen is a widely deployed hypervisor, KVM has quickly grown in popularity. Many people prefer KVM due to the ease of installation that comes from being a Type-2 VMM (i.e., it is integrated into the host operating system and can leverage the OS hardware support). Therefore, it makes sense to enable introspection capabilities for KVM. Moving in this direction is what motivated the library name change from XenAccess to LibVMI.

The original XenAccess software was built specifically for Xen – as the name implies. Therefore, function calls to interface with Xen were scattered throughout the code. In order to support KVM, we first refactored the code to contain all Xen-specific interactions in a single "driver". Next, we wrote a new driver to support KVM. And, finally, we setup LibVMI to dynamically determine which virtualization platform is available at startup; choosing the correct driver at that time.

The other piece of the puzzle was to actually access the VM memory for KVM. Unlike Xen, KVM does not provide any APIs to facilitate this access. So, to support LibVMI, we created a patch for KVM that enabled memory access through a unix domain socket. We used the libvirt library to gain the additional access that we needed (e.g., pausing and resuming the VM). Since the patch is somewhat challenging for users to deploy, we also enabled a technique to access memory through a KVM VM's GDB stub. GDB is the GNU Debugger. It provides a rich set of capabilities for viewing a running process or system. In this case, we could dump memory

through a GDB network protocol. But the resulting access is slower than using our KVM source code patch. Users can now choose between a harder to install software patch that provides faster memory access, and an easier to install GDB connection that provides slower memory access.

The end result is that LibVMI now support both Xen and KVM. Furthermore, it would now be very easy to write a driver to support another virtualization platform in the future.

2.2 API Improvements

The older XenAccess API required developers to manually map and unmap VM memory pages. This turned out to be arduous and error prone. Furthermore, developers often wrote code that abused this interface, resulting in large performance degradation. The new LibVMI API abstracts this low-level page mapping away from the developers and instead provides a more intuitive API based on familiar feeling read and write functions. Figure 4 shows a sampling of the current LibVMI API.

```
status_t vmi_init (vmi_instance_t *vmi, uint32_t flags, char *name)
status_t vmi_destroy (vmi_instance_t vmi)
addr_t vmi_translate_kv2p (vmi_instance_t vmi, addr_t vaddr)
addr_t vmi_translate_uv2p (vmi_instance_t vmi, addr_t vaddr, int pid)
addr_t vmi_translate_ksym2v (vmi_instance_t vmi, char *symbol)
addr_t vmi_pid_to_dtb (vmi_instance_t vmi, int pid)
size_t vmi_read_ksym (vmi_instance_t vmi, char *sym, void *buf, size_t count)
size_t vmi_read_va (vmi_instance_t vmi, addr_t vaddr, int pid, void *buf, size_t count)
size_t vmi_read_pa (vmi_instance_t vmi, addr_t paddr, void *buf, size_t count)
size_t vmi_write_ksym (vmi_instance_t vmi, char *sym, void *buf, size_t count)
size_t vmi_write_va (vmi_instance_t vmi, addr_t vaddr, int pid, void *buf, size_t count)
size_t vmi_write_pa (vmi_instance_t vmi, addr_t paddr, void *buf, size_t count)
void vmi_print_hex (unsigned char *data, unsigned long length)
unsigned long vmi_get_memsize (vmi_instance_t vmi)
status_t vmi_get_vcpureg (vmi_instance_t vmi, reg_t *value, registers_t reg, unsigned long
vcpu)
status_t vmi_pause_vm (vmi_instance_t vmi)
status_t vmi_resume_vm (vmi_instance_t vmi)
void vmi v2pcache flush (vmi instance t vmi)
```

Figure 4 Sampling of the current LibVMI API. The complete API contains a variety of additional convenience functions designed to make VMI development easier.

The read and write memory functions transparently handle page boundaries so that the VMI developers can focus more on their programming task. One side effect of this new API is that LibVMI must internally manage mapping and unmapping pages. It must do this in a smart way so as to achieve good performance. Section 2.3 describes the page cache and other caching techniques.

Other useful functions include those that read the current CPU registers, pause the VM, and resume the VM. There are also a series of functions for managing the various LibVMI caches. Figure 3 shows two of these functions as an example. These manage the virtual to physical address cache by allowing developers to manually add mappings and flush the cache.

2.3 Performance

LibVMI is built around the seemingly simple concept of accessing memory in the VM. This operation is actually quite costly. Therefore, naïve VMI applications tend to be very slow. LibVMI solves this problem by managing four different caches inside the library. Three caches handle basic mappings: virtual address to physical address, virtual address to process identifier, and kernel symbol to virtual address. The fourth cache is the page cache.

The page cache handles mapping or copying VM memory pages into memory as needed to support LibVMI's memory read and write functions. Our first attempt at designing the page cache was to use a hash table. The hash value would be derived from the physical page address. And the hash table entry could provide meta-data about the memory page, along with a pointer to the data itself. This worked, but the performance was slower than expected because the hash table grew quickly, resulting in increasingly slow hash table lookups.

We solved this problem by maintaining a second data structure. This structure is a list that contains the hash values in the order accessed. Specifically, the front of the list has the hash value associated with the most recently accessed page. And the end of the list has the hash value associated with the least recently accessed page. When the list grows beyond a predetermined size (experimentally we determined 512 entries was a good size, balancing the performance tradeoffs), the second half of the list is removed. We also remove the related entries from the hash table at the same time. This ensures that the freshest entries remain in the hash table, while still providing loose management for the overall hash table size. Figure 5 provides a graphical representation of this page cache algorithm.

The end result of this new page cache algorithm was a significant improvement in performance. Figures 6 and 7 show the before and after performance numbers for LibVMI. Noting the logarithmic scale, we see a two orders of magnitude speedup in one of our test cases. The graphs show the vmi_read_pa API call being used in two different ways. The first is 1875 calls to read 4 bytes each. The second is 1 call to read 7.5k bytes. Each call results in a final result of reading 7.5k bytes, but they exercise the various cache algorithms within LibVMI differently. Ideally, with all of the cache enabled, both calls should perform similarly. Our results show that we achieved this goal. This means that VMI developers can focus less on the performance impact of their code and more on the actual memory analysis tasks. Therefore, we feel that this result is an important step towards simplifying VMI software development.

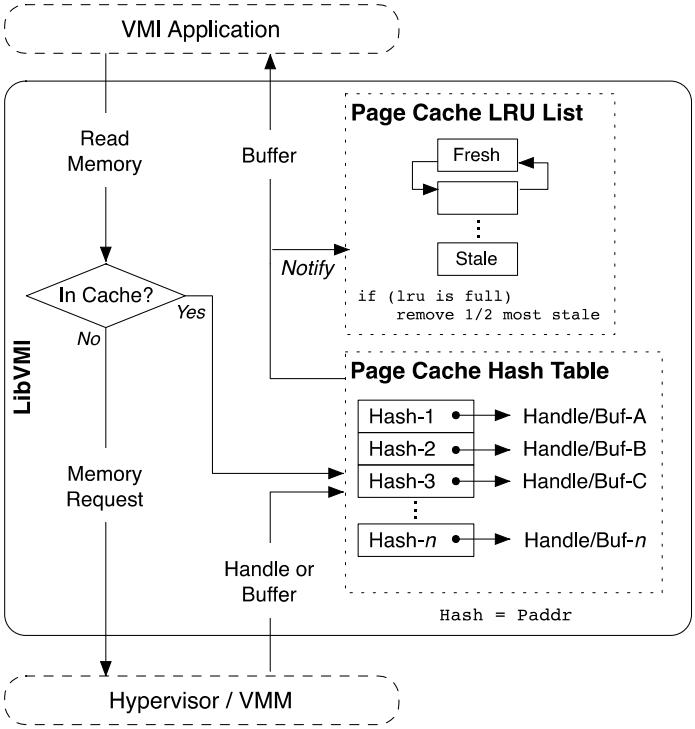

Figure 5 LibVMI's page cache algorithm balances memory overhead with performance, ensuring that the related data structures never grow too large.

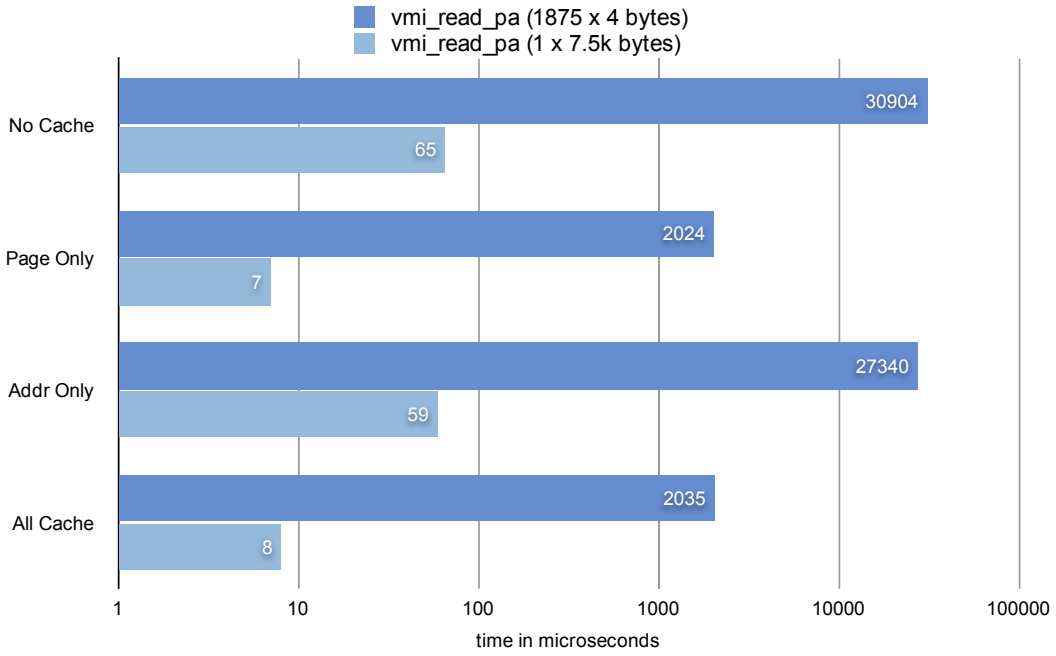

Figure 6 LibVMI performance before our cache optimizations.

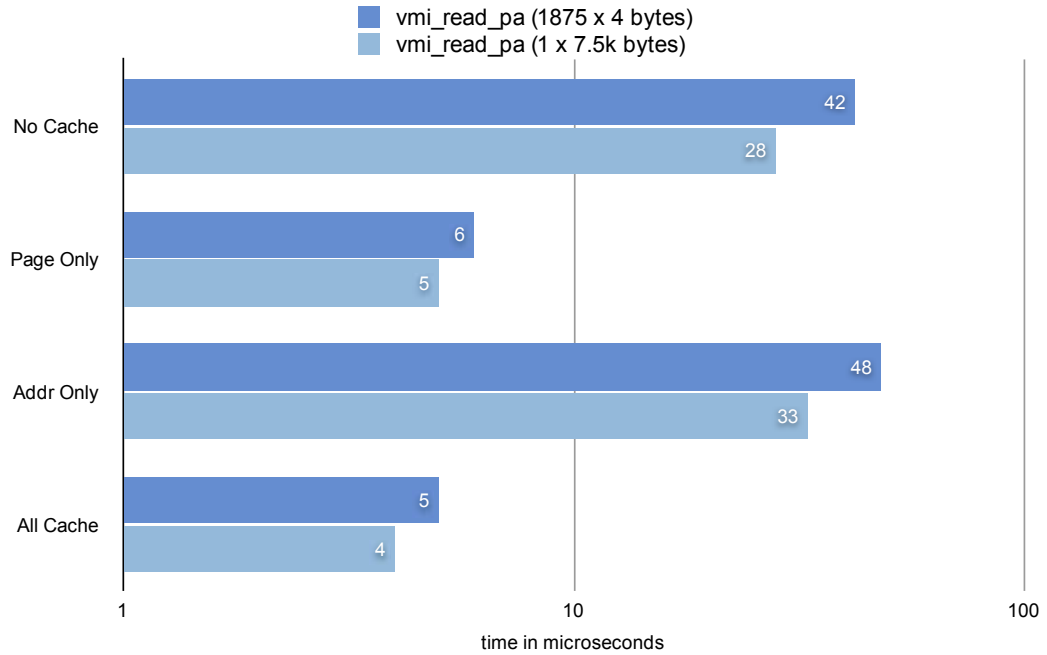

Figure 7 LibVMI performance after our cache optimizations.

2.4 64-bit Guest Support

XenAccess supported viewing the memory of 32-bit guests. However, as 64-bit guests are becoming increasingly popular, we decided to extend LibVMI to support 64-bit guests as well. In order to understand the work involved for this addition, let's step back and consider how LibVMI handles a basic read API function call.

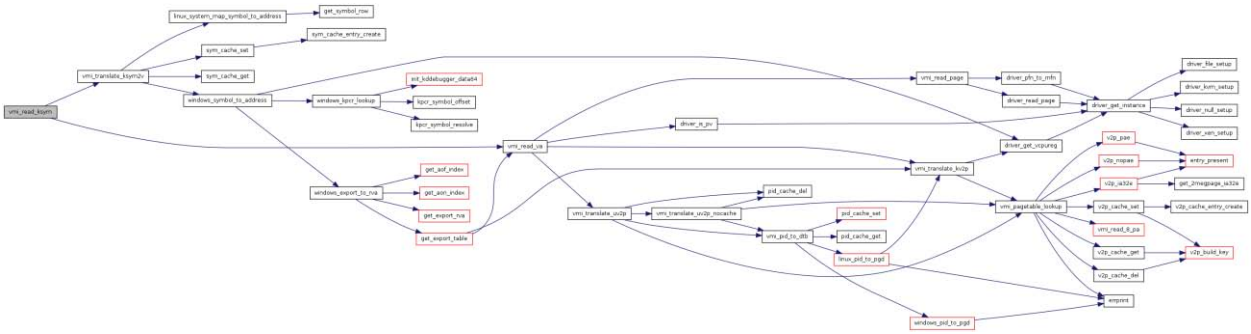

Figure 8 While the details in this picture are too small to read, the key point is that significant work happens for each API call. In this case, the vmi_read_ksym call must handle reads around page boundaries, resolve the kernel symbol, translate the kernel symbol to a physical address, and perform the actual read from the VMM.

Figure 8 shows the internal function calls performed by LibVMI for a call to the vmi_read_ksym function. To support 64-bit guests, minor changes were needed in the PE parsing code that converts a Windows kernel symbol to a virtual address. On the other hand, major changes were required to support translating a 64-bit virtual address to a physical address. This required implementing the 64-bit address translation algorithms as described in the Intel Architecture Manuals. Finally, we had to ensure that all variables representing addresses throughout LibVMI could handle 64-bit values.

3. PYVMI AND VOLATILITY INTEGRATION

LibVMI is written in the C programming language. This is useful for many applications, while also enabling the best possible performance. However, many memory analysis tools are now beginning to use the Python programming language instead of C. Python enables rapid prototyping and contains some language features that are particularly useful for VMI developers, such as the ability to control the semantics of attribute access through constructs like object.__getattr__(self, name).

Perhaps the most popular memory analysis framework today is Volatility. Volatility is written in Python for the reasons discussed above. Also, effective with Volatility 2.0, it supports pluggable address spaces. This means that the mechanism that Volatility uses to acquire memory data can be easily extended to support other memory capture techniques.

In order to leverage both the growing use of Python and the rich memory analysis capabilities of Volatility, we extended LibVMI with a feature complete Python wrapper and a Volatility address space.

3.1 PyVMI: A Python Wrapper for LibVMI

Python has native support for extending the language with C libraries. Therefore, writing a Python wrapper for LibVMI was straightforward. Each function in the external LibVMI API has a semantically equivalent function in PyVMI. In addition, we added a zread(…) function in PyVMI. This function will read from the desired location, but will never fail. Any bytes that can't be read will simply be replaced with zeros. While generally useful, the motivation for adding this specific function was to facilitate improved integration with Volatility. Figure 9 shows the updated software stack with PyVMI.

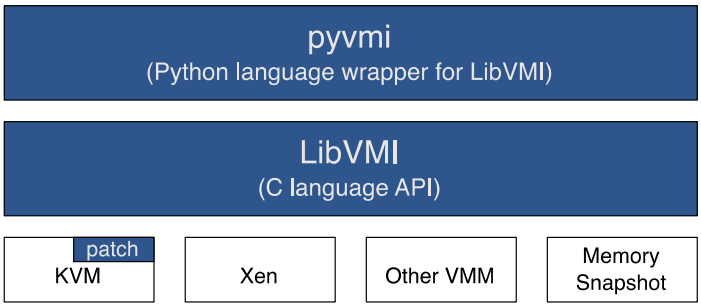

Figure 9 Software stack with PyVMI wrapper on top of the C language LibVMI library. The portions in blue represent code written for this project.

3.2 PyVMI Address Space for Volatility

Volatility has an active development community. This community has created many rich memory analysis capabilities. Volatility is designed as an extensible framework. Developers extend Volatility by writing plugins that perform new memory analysis tasks. Some existing plugins perform tasks such as listing the running processes, listing the open network connections, listing the handles for each process, parsing the Windows registry, displaying VAD tree

information, and showing the SSDT entries. The actual list of available plugins is long and grows quickly due to the strong development community.

Volatility is designed to work on forensic memory snapshots. In this mode, a forensic analyst would take a physical memory image from a target machine, and then use Volatility to extract useful information from that image. However, since Volatility already contains significant information on the Windows memory layout and because Volatility greatly simplifies the development of memory analysis tools, we wanted to integrate Volatility with LibVMI to facilitate analysis on a running virtual machine.

Thanks to Volatility's address space plugins, this was a straightforward task. We wrote an address space plugin that enabled Volatility to use PyVMI for its physical memory access. The end result is the software stack seen in Figure 10.

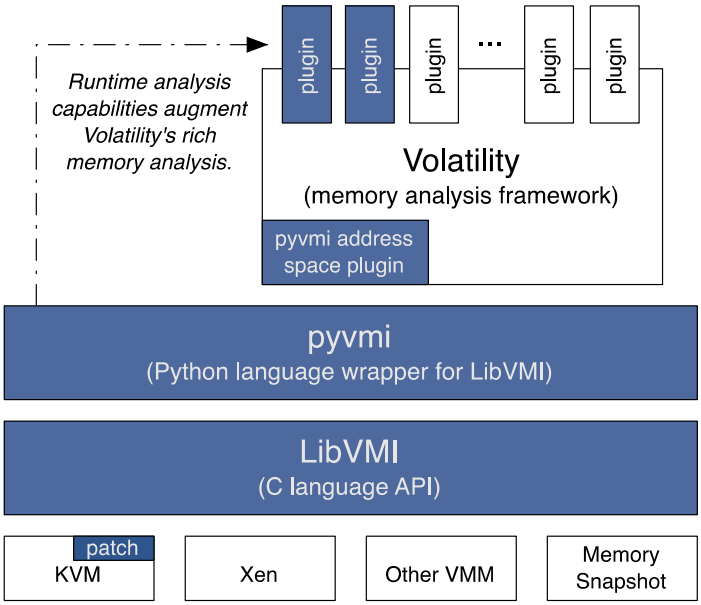

Figure 10 Software stack with Volatility address space plugin. The portions in blue represent code written for this project.

Now all of the plugins available for Volatility will work on a running virtual machine. And, it is possible to leverage these plugins when designing new VMI applications. We hope that this will help tie together the creative contributions from both the forensic memory analysis community and other runtime / dynamic analysis communities (e.g., malware analysis).

.

4. FUTURE WORK

LibVMI as it exists today is very usable for a variety of VMI development tasks. However, there are always areas for improvement. This section explores some of the best steps for improving LibVMI by making it more capable and easier to use.

Hypervisors are starting to include event notifications. In the case of Xen, this includes notification when a marked page in memory has been read, written, or executed. It also includes notification when certain VCPU registers change. We would like to bring this functionality up into LibVMI as it is a natural complement to the existing introspection capabilities. We would also like to include this functionality in PyVMI to allow Volatility plugins to benefit from events.

The Volatility and KVM integration are both in the early stages. More work is needed to remove bugs and improve performance in these components.

Finally, it would be useful to have richer symbol access support. Currently LibVMI provides access to Windows kernel symbols (found through memory analysis) and Linux kernel symbols (found in the System.map file). We would like to extend this to include user-level process symbols and to locate the Linux kernel symbols in memory, which should improve performance.

5. CONCLUSIONS

LibVMI evolved from the XenAccess project. Throughout the course of this LDRD, we improved LibVMI in a variety of ways with the overall goal of making VMI development easier. We succeeded in accomplishing this goal through API improvements to the library, support for other virtualization platforms, support for 64-bit guests, performance improvements, and integration with the Volatility memory analysis framework.

At the time of this writing, we just released LibVMI Version 0.8. This software is freely available under the GNU Lesser General Public License (LGPL). For more information, please see the project website at http://code.google.com/p/vmitools/.

A1. DISTRIBUTION

Qty	Mail Stop	Name	Org
1	MS 0359	Donna Chavez, LDRD Office	1911
1	MS 0621	Tan Thai	5630
1	MS 0621	Roxana Jansma	5631
1	MS 0899	Technical Library	9536 (electronic copy)
1	MS 1231	Anthony Thornton	5220

Sandia National Laboratories