Libvmi:A Library for Bridging the Semantic Gap between Guest OS and VMM

Haiquan Xiong[1,2] Zhiyong Liu[1] Weizhi Xu[1,2] Shuai Jiao[1,2]

{xionghaiquan, zyliu, xuweizhi, jiaoshuai}@ict.ac.cn
1. State Key Laboratory of Computer Architecture, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China
2. Graduate University of Chinese Academy of Sciences, Beijing, China

***Abstract*—Semantic gap is one of the most important problems in the virtualized computer systems. Solving this problem not only helps to develop security and virtual machine monitoring applications, but also benefits for VMM resource management and VMM-based service implementation. In this paper, we first review the general architecture of virtual computer systems, especially the interaction principles between Guest OS and VMM so as to better understand the causes of semantic gap. Then we consider how to build a library that can integrate the commonalities existing in different VMMs and Guest OSes. It should be general enough to facilitate the above mentioned applications development. In order to more clearly illustrate this issue, we select Libvmi as the case study, elaborating its design philosophy and implementation. Finally, we describe how to use Libvmi with two application examples and verify its correctness and effectiveness.**

Keywords-Semantic Gap; Libvmi; Guest OS; VMM; Virtualization; KVM; Xen

I. INTRODUCTION

Virtualization technology has become more and more important in the computer industry both to improve security and to support multiple efficient virtual machines on the same hardware platform. It is widely used in server consolidation, systems software developing and debugging, fault tolerance, system security, workload balancing and etc.[1]. In order to make use of virtualization technology further, there are still many problems to be solved. The semantic gap is just one of the most important. It refers the lack of higher-level knowledge of Guest OS within the VMM[2].

The semantic gap is a result of the independent design of Guest OS and VMM. On the one side, Guest OS gets resources through the virtual interface the same as it gets from the real machine. It cannot get aware that it is running on a VMM, and cannot do the effective adjustments or adaptations to improve performance. On the other side, the VMM mainly provides a logically identical or similar virtual computing environment for Guest OS and can only see the low level raw activities without getting much of the semantic meaning hidden inside. For example, a VMM is not inherently privy to the semantics of a Guest OS's basic abstractions (*e.g.*, processes, address space), its policies, or its performance goals. Currently, no standard interfaces exist that allow a VMM to query a guest operating system for these details.

Bridging the semantic gap is not only important for security application, but also has great benefits for VMM resource management and VMM-based service implementation. In security space, to perform useful analysis of the VMs, VMMs must deal with the semantic information gap between the low-level activities available to them and the high-level Guest OS abstractions they need[3; 4]. For example, in x86 architecture, when a low-level activity like CR3 value changing event occurs, combining with the Guest OS process address space semantic knowledge, VMM can identify process switching behaviors inside Guest OS. Besides，semantic gap bridging can also help VMM make better decisions on resource scheduling and provide more opportunities for VMM-based service implementation because VMMs have more information about Guest OSes. For instance, in[5], the author makes use of Antfarm for xen to distill the Guest OS process semantic information for implementing anticipatory scheduling in VMMs and obtain significant disk throughput improvements. [6] explores the Guest OS memory semantic information hidden in the low level events such as page table updates, page faults and etc. to estimate VMs' memory dynamic demands, then uses it for memory balancing in virtual computing environment.

In prior works, researchers have developed many security applications with virtual machine introspection[3] and VMM resource management optimizations with Guest OS semantic information exploitation[5-7] . Each of these applications has one thing in common: they each need to monitor the low level interaction events between Guest OS and VMM, and derive the specific information. However, the underling mechanics of the interactions between them and how to design a general library for bridging the semantic gap have not been adequately addressed in the literature.

In order to help understand and solve this problem, in this paper we first review the general model of virtual computer systems and the interaction principles between Guest OS and VMM to better understand the causes of semantic gap. Then we consider whether there is a possibility to build a general library for bridging the semantic gap and what basic requirements it should meet. Then we select Libvmi as the case study to show how such a library can be designed and implemented. Finally, with two application examples: Processes Listing and Kernel Module Listing, we describe how to use Libvmi and verify its correctness and effectiveness.

The remainder of this paper is structured as follows: in section II, some virtual computer system background

978-0-7695-4858-6/12 $26.00 © 2012 IEEE
DOI 10.1109/CIT.2012.119

knowledge is reviewed. Section III describes the design and implementation of Libvmi library. In section IV, we do two experiments to illustrate Libvmi's use and verify its correctness. Section V discusses some related work, and finally in section VI concludes this paper.

II. VIRTUAL COMPUTER SYSTEMS

To help better understand the following contents, it is beneficial for us to review the common architecture of virtual computer systems, especially some basic knowledge about the interactions between Guest OS and VMM.

Generally, a virtual computer system (VCS) consists of a perfect multiplexing of the real physical machine. This multiplexing includes the scheduling of the CPU, but also the sharing of the memory and of the other resources.

From Figure 1, it can be seen that a VCS typically includes the following components: the virtualizable hardware, VMM and some VMs including their Guest OS kernels and applications. Below we describe them each, including the interactions between Guest OS and VMM.

A. Virtualizable Hardware Architecture

In the classical virtualization theory paper[8], the author establishes a simplified model of third generation architectures and, from which propose a theorem on what requirements an architecture should meet to virtualize. It can be simply described as follows: For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that architecture is a subset of privileged instructions.

The theorem can be used to verify whether an existing architecture can support a classical VMM, if not, why and how to add virtualization extensions to make it virtualizable. For example, on the basic rules founded by the theorem, [9]and[10] separately analyze Pentium and IA-64 architecture and identify the virtualizable problem instructions, then by adding the virtualization extensions such as Intel VT-x and AMD SVM, both of them make the IA architecture classically virtualizable.

Besides, the principles embodied in the virtualizable architecture can also help to explore the mechanisms for deriving VMs internal information within VMM or other VMs[4]. And most important, it makes us recognize the fact that any resource sensitive actions can be intercepted in some way, so its effects can be flexibly controlled by software.

B. Virtual Machine Monitor

Upon the virtualizable hardware sits the Virtual Machine Monitor (VMM). It is a thin system software that provides virtual hardware interfaces that resemble physical hardware. Operating system, which traditionally runs on real machine, can run on the virtual hardware.

Although VMMs have different implementation styles such as hypervisor or hosted architecture, they all have similar logical structures. Generally, every VMM consists of three basic components: dispatcher, allocator and interpreter. You can see them at the bottom half of Figure 1.

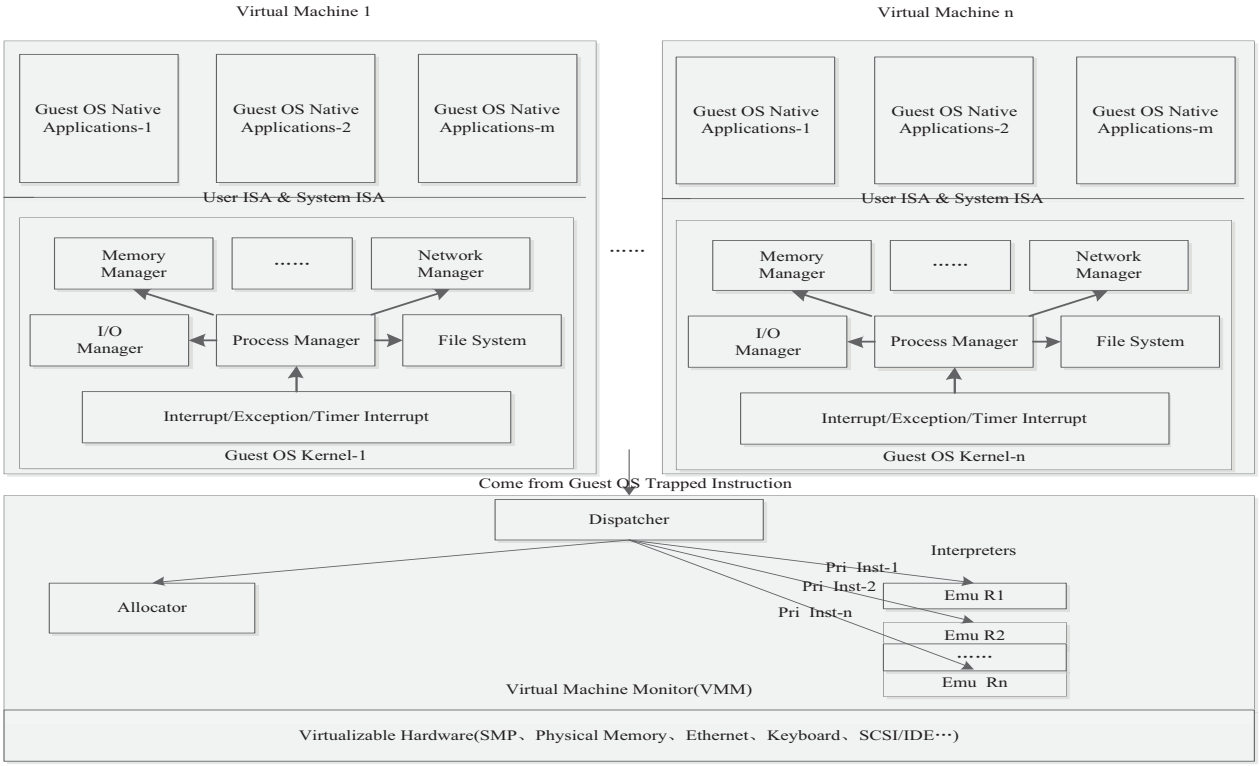

Figure 1 A General Virtual Computer System

Dispatcher is the first entry when a running Guest OS traps into VMM due to sensitive privileged instruction. It is the top level control module that decides ether to call allocator or interpreter routings.

Allocator decides what system resource(s) are to be provided. In the case of a single *VM,* the allocator needs only to keep the VM and the VMM separate. In the case of a VMM which hosts several VMs, it is also the allocator's task to avoid giving the same resource (such as part of memory) to more than one VM concurrently. The allocator will be invoked by the dispatcher whenever an attempted execution of a privileged instruction in a virtual machine environment occurs which would have the effect of changing the machine resources associated with that environment.

Interpreters are the ultimate routing executed for the Guest OS privileged instruction emulation. The purpose of each of the routines is to simulate the effect of the instruction which is trapped. So we can conclude that every time the Guest OS encounters a trap event, there must exist a corresponding routing for them. In practice, it help us find the specific source code routings in real VMMs and modify them accordingly.

C. Virtual Machines and Operating Systems

The VMM logically divides a real physical machine into many virtual machines that can host operating systems. The core task of operating systems is to provide their programs an isolated execution environment. Each of these programs feels that it occupies the entire machine itself, that is to say, it has its own CPU, its own memory, its own I/O devices and so on. In order to realize the illusion, operating systems create a series of software abstractions such as process, memory address space, I/O device model, virtual file system, unified buffer cache and etc. Among them, the process abstraction is the key concept, because it provides the connections to many of the fundamental abstractions that a program relies on, like private address spaces, and serves as the basic resource container and security isolation boundary for user tasks. The process implements a running environment with the help of other software abstractions it connects. All of them work together to complete the dynamic resource mapping needed by the program being serviced. The entire system relies on interrupts, exceptions, timer interrupts, mmu and any other hardware mechanisms to realize resource time-sharing or space-sharing. The events generated by these mechanisms can trigger process context switching, which logically changes to a new set of resource mapping functions for a new program.

D. The Interaction between Guest OS and VMM

After discussion of both Guest OS and VMM, we consider the interactions between them. For the purpose of cleaner description, first we introduce some resource mapping essentials.

Here we define the set of process virtual resource names V= {V0,V1,…,Vj} to be the set of names addressable by a program executing. Let P = {P0, P1,... , Pn } be the set of physical resource names of a virtual machine and finally let R = (R0, R1, ..., Rm) be the set of real resources of the hardware. The Guest OS implements the first mapping which is from V to P, while VMM is in charge of the second mapping which is from P to R.

The first mapping is the same as the traditional operating system, so we do not make further discussion. We mainly focus on the VMM resource mapping. As the above mentioned, an operating system create an isolated execution environment for an application by a series of software abstractions. These abstractions can be roughly divided into process abstractions, memory abstractions, I/O abstractions and other miscellaneous abstractions. Guest OS implements the resource mapping using its software abstractions which interact with the VMM. Every abstraction has its corresponding sensitive privileged instructions which trap when executing. Theoretically, VMMs can see all the events occurred in Guest OSes. However, the semantic information behind the scene is not easy to extract because VMMs have little knowledge about the association between low level observable events and high level Guest OS software abstractions. In order to facilitate solving the problem, VMMs often come with some low level control library. It corresponds to Semantic Gap Bridging Control Module in Figure 2.

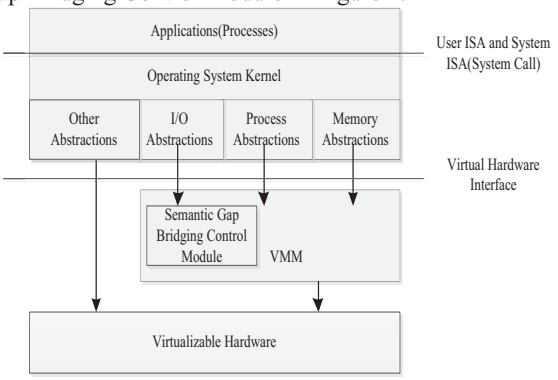

Figure 2 The Interaction between Guest OS and VMM

For example, in Xen, there exists an libxc which encapsulates the services implemented in /proc/xen/privcmd, /dev/xen/evtchn and /dev/xen/gntdev, providing an ioctl interface for applications to communicate with Xen. While in the Qemu&KVM, there exists another library libvirt which in internally uses QMP(Quick Monitor Protocol) to communicate with Qemu&KVM virtual machines.

III. THE LIBVMI LIBRARY

A. The Architecture Design Philosophy

1) The Purpose of Libvmi Library

The primary goal for the Libvmi[11] library is to illustrate how to distill the commonalities existing in the interactions between Guest OS and VMM and use them for bridging the semantic gap problems. So a lot of security application can effectively use the library with less engineering efforts.

2) Basic Ideas and Design Philosophy

Every Guest OS running in the virtual machine gets its resources through the virtual interface. In theory, all Guest OS behaviors and internal state can be obtained within the VMM because it is the primary and direct hardware resources manager in the virtualized computing system. To develop applications that need virtual machine states, such as processor registers, memory, disk, network, and any other hardware-level events, it is efficient to consider building a general purpose library. This library needs to make full use of the commonalities of VMMs and Guest OSes. Specifically, it should take into account the following basic requirements: 1, Less or no modifications to the VMMs; 2, No modifications to the VM or the target OS so it is more applicable. 3, It should assist rapid application development, this requires that the library APIs be easy to use. 4, The library should be general and extensible to include any other possible aspects beneficial for bridging the semantic gap so that more and more information about the target Guest OS can be obtained or derived.

Figure 3 shows the overall software architecture, Libvmi sits between monitor applications and VMM specific low level semantic library, for example, libxc for Xen and libvirt for Qemu&KVM. It can include service routings for accessing memory, disk, network, CPU registers and any abstractions of Guest OSes.

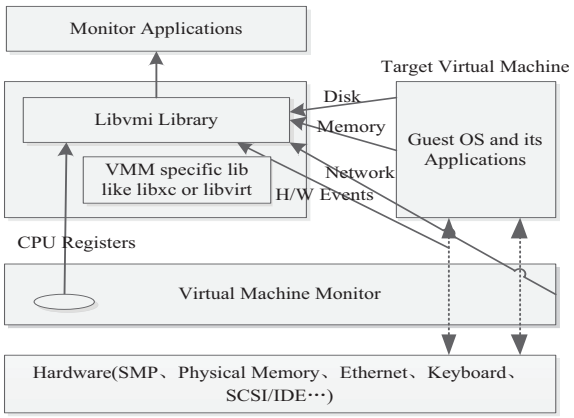

Figure 3 The Overall Libvmi Architecture

B. Implementation

Libvmi[11] library is implemented in C as a shared library. It makes use of libxc in Xen and libvirt in Qemu&KVM. The current version works with Xen (v3.x through 4.1) and KVM (with patch against QEMU-KVM 0.14), however the design techniques here can be extended to work with other VMMs and OSes. The design philosophy resembles the traditional virtual file system in Linux, see Figure 4. First, the top Libvmi API is similar to file system call API, and Driver Interface is equivalent to virtual file system common interface, finally in the below are the specific VMMs with their corresponding low level library drivers, such as libxc with Xen and libvirt with KVM. In the following, we will describe the implementation details of Libvmi.

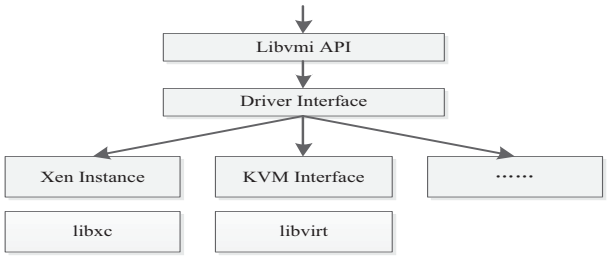

Figure 4 The Libvmi Layered Implementation structure

1) Libvmi API

On the whole, the Libvmi APIs can be divided into two categories: one is the library initializing and destroying management APIs, the other is utility APIs for different kinds of purposes, such as memory accesses APIs, disk accesses APIs, CPU registers accesses APIs and etc.

a) Libvmi Library Initializing and Destroying API

- **vmi_init**: Initializes access to a specific virtual machine given a name. All calls to vmi_init must eventually call vmi_destroy. This is a costly function in terms of the time needed to execute. You should call this function only once per virtual machine, and then use the resulting instance when calling any of the other utility library functions.
- **vmi_destroy** : Destroys an instance by freeing memory and closing any open handles.

b) Libvmi Utility API Examples

For brief purpose, here we select a few typical memory access utility library APIs as examples.

- **vmi_read_addr_ksym** : Read a specified kernel symbol value from a target virtual machine, inside, the kernel symbol will be converted to a virtual address by looking up the system symbol map.
- **vmi_read_addr_pa**: Reads count bytes from memory located at a specified physical address in a target virtual machine and stores the output in a buffer.
- **vmi_read_addr_va**: Reads count bytes from memory located at a specified virtual address in a target virtual machine and stores the output in a buffer.
- **vmi_write_addr_ksym** : Writes a specified kernel symbol value to a target virtual machine.
- **vmi_write_addr_pa**: Writes count bytes to memory located at a specified physical address in a target virtual machine from a buffer.
- **vmi_write_addr_va**: Writes count bytes to memory located at a specified virtual address from a buffer.

Above we only give an overview of some of the Libvmi APIs. In principle, the others follow the same design ideas. That is, the library can be extended to include disk, network and the like in a similar way.

2) Driver Interface

Libvmi inherits the design philosophy from the virtual file system. Driver Interface abstracts the common VMM and Guest OS features and application requirements such as the connection methods, the VM pausing and resuming methods, memory accessing methods and the like. Figure 5 gives the driver interface definition. It is a structure composed of various common function pointers substituted by specific VMM driver functions.

```
struct driver_instance{
    status_t (*init_ptr)(vmi_instance_t);
    void (*destroy_ptr)(vmi_instance_t);
    unsigned long (*get_id_from_name_ptr)(vmi_instance_t, char *);
    unsigned long (*get_id_ptr)(vmi_instance_t);
    void (*set_id_ptr)(vmi_instance_t, unsigned long);
    status_t (*get_name_ptr)(vmi_instance_t, char **);
    void (*set_name_ptr)(vmi_instance_t, char *);
    status_t (*get_memsize_ptr)(vmi_instance_t, unsigned long *);
    status_t (*get_vcpureg_ptr)(vmi_instance_t, reg_t *, registers_t,
unsigned long);
    status_t (*get_address_width_ptr)(vmi_instance_t vmi, uint8_t *
width);
    void *(*read_page_ptr)(vmi_instance_t, addr_t);
    status_t (*write_ptr)(vmi_instance_t, addr_t, void *, uint32_t);
    int (*is_pv_ptr)(vmi_instance_t);
    status_t (*pause_vm_ptr)(vmi_instance_t);
    status_t (*resume_vm_ptr)(vmi_instance_t);
};
```

Figure 5 Libvmi Common Driver Interface

3) Specific Virtual Machine Monitor Driver

a) Xen

```
typedef struct xen_instance{
    libvmi_xenctrl_handle_t xchandle;
    unsigned long domainid;
    int xen_version;
    int hvm;
    xc_dominfo_t info;
    uint8_t addr_width;
    struct xs_handle *xshandle;
    char *name;
} xen_instance_t;
```

Figure 6 Xen Specific Driver Interface

Figure 6 shows the specific definition for Xen virtual machine instance. The most important field xchandle(handle to libxc library) is used to create a connection with a Xen virtual machine instance.

b) KVM

```
typedef struct kvm_instance{
    virConnectPtr conn;
    virDomainPtr dom;
    unsigned long id;
    char *name;
    char *ds_path;
    int socket_fd;
} kvm_instance_t;
```

Figure 7 KVM Specific Driver Interface

Figure 7 shows the specific definition for KVM virtual machine instance. The fields conn and dom is used to create a connection with KVM virtual machine instance.

4) The Libvmi Library Initializing Flow

Figure 8 gives the Libvmi library basic initializing flow. First, the system allocates a vmi instance variable and initializes it with zero value, uses the input parameters to select the specific VMM driver such as Xen, KVM and etc. Then, it continues to obtain the target Guest OS configuration, memory size, paging mode and finally completes vmi instance variable initializing. After all things done, applications can use any Libvmi utility APIs.

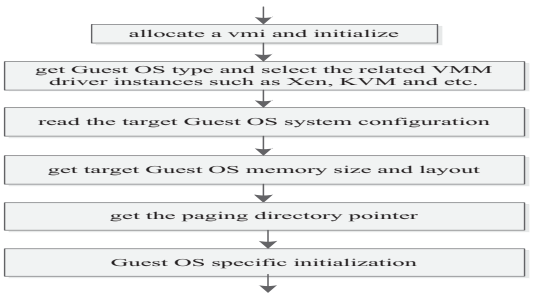

Figure 8 The Libvmi Initializing Flow

IV. EXPERIMENTS AND APPLICATION EXAMPLES

This section gives the basics of using Libvmi and presents two application examples: Guest OS Processes Listing and Kernel Modules Listing, illustrating Libvmi's usefulness and its correctness. The experimenting system is based on a 2.00 ghz T7200 Intel Core 2 Duo dual core machine with 4 GB of memory. The Xen version is 4.0.1, Domain 0 and Domain U separately use Linux kernel 2.6.32.39 and 2.6.35.8. And both of them use the same root file system from fedora 13. Besides, all experiments are based on x86-32 architecture.

A. The Basics of Using the Libvmi Library

1) Target Guest OS Configuration

Modern operating systems vary in implementation based on the hardware architecture upon which they run and other specific details, but many share similar logical structure and software abstractions, such as process, memory address space, paging table and so on. All operating systems have a few core data structures and global variables, through them plus the target Guest OS internal semantic knowledge, more useful and hidden information can be easily exploited and distilled. For example, the common structures include process control structure, memory management description, page table directory and etc. According the application common requirements, we can select the most widely used from the above mentioned core data structures and variables as the base anchors or keys, each of these has the following "key=value" format.

```
<VM Name> {
        <key> = <value>;
        <key> = <value>;
        }
```

The VM name is what appears when you use the 'xm list', 'xl list', or 'virsh list' commands. Currently, there are 9 different keys available for use. The ostype is used by both

Linux and Windows targets. The sysmap is only used for Linux targets. The others specify offsets such that the linux * values that are required for Linux and the win * values that are required for Windows. The available keys are listed below:

- sstype: Linux or Windows guests are supported.
- sysmap:The path to the System.map file for the VM. Note that this file must be copied into the dom0 or host VM from the domU or guest VM so that Libvmi can access it.
- linux tasks: The number of bytes (offset) from the start of the struct until task struct->tasks from linux/sched.h in the target's kernel.
- linux_mm: Offset to task_struct->mm.
- linux_pid: Offset to task_struct->pid.
- linux_pgd: Offset to mm_struct->pgd.
- win tasks: Offset to EPROCESS->ActiveProcessLinks.
- win pdbase: Offset to EPROCESS->Pcb->DirectoryTableBase.
- win_pid: Offset to EPROCESS->UniqueProcessId.

Here are two examples: One is for Linux in Figure 9, the other is for Windows in Figure 10.

```
Fedora13 {
   ostype = "Linux";
   sysmap = "kernel symbol table";
   linux_name = 0x30c;
   linux_tasks = 0x1d8;
   linux_mm = 0x1f4;
   linux_pid = 0x214;
   linux_pgd = 0x28;
   linux_addr = 0x84;
}
```

Figure 9 Linux Configuration

```
Windows-XP {
   ostype = "Windows";
   sysmap = "windowsxp.txt";
   win_tasks  = 0x88;
   win_pdbase  = 0x18;
   win_pid    = 0x84;
}
```

Figure 10 Windows Configuration

These key configuration values can be obtained by programming a specific kernel module or directly from the real configuration in Linux systems; While on the part of Windows, it can also be obtained by kernel debugging tools, the details refer. [12]

2) Application Template

The logical structure of applications using the Libvmi library first includes "libvmi.h" header file and use vmi_init() function to initialize an vmi instance structure which holds information that is used throughout the executing process. Any work that can be done "up front" and cached is held in this structure. This includes locating the address of the kernel page directory, initializing a handle to libxc or libvirt, initializing a pointer to a PFN to MFN lookup table, determining if the VM is paravirtualized or fully virtualized, and etc. Once completed, the application uses the Libvmi utility APIs to implement its own specific purpose. Finally, a call should be made to vmi_destroy() to free any memory associated with the vmi instance structure. Figure 11 shows the general source code template.

```
/*header file include area*/
int main (int argc, char **argv)
{
   vmi_instance_t vmi;
   /* command line parameters preprocessing*/
    /* initialize the libvmi library */
    vmi_init(vmi)
   /* pause the guest os for consistent memory access */
    vmi_pause(vmi);
    /*application specific processing*/
   ……
   /* resume the guest os */
   vmi_resume(vmi);
   /* cleanup any memory associated with the framework libray */
   vmi_destroy(vmi);
   return 0;
}
```

Figure 11 The Template for Applications Using Libvmi

B. Guest OS Processes Listing

Stealth rootkit techniques can be used to hide long-lived malicious processes in operating systems. How to detect and identify this situation in target operating system is key to system security. As for this problem, virtualization technologies provide a novel solution. Based on the Libvmi library, we can get the target Guest OS processes list that cannot be bypassed by the traditional techniques. At the same time, we also can get another view of the current processes in the target Guest OS utilizing its native tools, such as ps in Linux and tasklist in Windows or the like. Through the comparisons of the two view listings, it can derive whether the target Guest OS has hidden processes. In the following, we talk about the basic principles, algorithms and show the experimental results.

1) Basic Principles and Algorithms

Linux, as any other Unix-like operating systems, after completing all the basic hardware initialization and filling all the resource management tables, it uses the static **init_task** struct variable as the first process template and sets the required values to make it runnable. Thereafter, all other processes are modeled after it and linked. Finally all lively processes are organized in a double- linked list shown as in Figure 12.

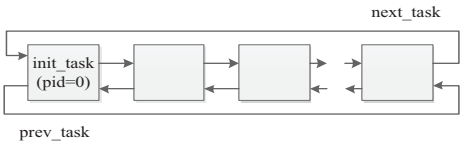

Figure 12 Linux Process Double Link List

Based on the above knowledge, to get the target Guest OS processes listing, it first needs to get the variable **init_task**, the header of process double-link list, then by travelling, all other processes can be obtained. Figure 13 give the basic Guest OS processes enumeration algorithms and Figure 14 lists some core program segments.

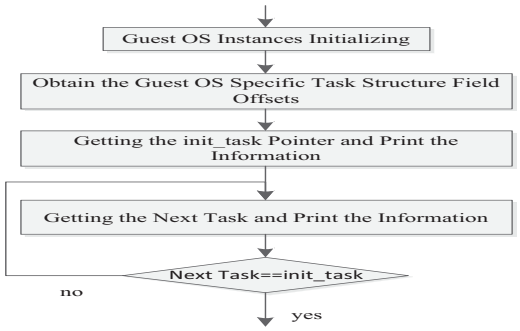

Figure 13 Guest OS Processes Enumeration Algorithms

```
1,    /* initialize the libvmi library */
        vmi_init(…)
2,    /* geting the offset values */
        tasks_offset = vmi_get_offset(vmi, "linux_tasks");
        ……
3,    /* get the head of the list */
        addr_t init_task_va = vmi_translate_ksym2v(vmi, "init_task");
        vmi_read_addr_va(vmi, init_task_va + tasks_offset, …);
        printf("[%5d] %s\n", pid, procname);
        list_head = next_process;
4,    /* walk the task list */
      while (1){
        /* follow the next pointer */
       addr_t tmp_next = 0;
        vmi_read_addr_va(vmi, next_process, 0, &tmp_next);
        /* if we are back at the list head, that is init_task, we are done
*/
5,    if (list_head == tmp_next){
          break;
        }
        /* print out the process name */
        procname = vmi_read_str_va(vmi, next_process + …);
        vmi_read_32_va(vmi, next_process +…);
        ……
        next_process = tmp_next;
  }
```

Figure 14 Guest OS Processes Listing Core Program Fragments

2) Experimental Results

In Table 1, it is divided into two columns: Host OS and Guest OS. The Host OS column represents the result gotten with Libvmi library, while Guest OS column represents the result directly obtained from the target Guest OS native tools. We can see that almost the same processes listing on both Host OS and Guest OS except for a few differences; it is because of the watching time differences between them. It does not affect its usefulness for some security application just like the above mentioned hidden processes detection.

Table 1 The Processes Listing Obtained from Host OS and Guest OS

Host OS	Guest OS
Process listing for VM Fedora13 (id=1) [1] init [2] kthreadd …… [1594] gconf-helper …… [2011] sleep **[2012] vim**	PID TTY TIME CMD 1 ? 00:00:00 init 2 ? 00:00:00 kthreadd …… 1594 ? 00:00:00 gconf-helper …… 2001 ? 00:00:00 sleep **2002 pts/0 00:00:00 ps**

C. Guest OS Kernel Modules Listing

1) Basic Principles and Algorithms

In the virtualized computing environment, some time we need to view what kernel modules Guest OS have loaded, especially when we want to inject kernel module to Guest OS from VMM[13], so here we give another example for listing the Linux kernel modules.

The principles and algorithms are similar to the Guest OS Processes Listing. There is a struct module global pointer variable named **modules**. When a new kernel module is loaded into the Linux kernel, it will add to the list and finally form a double linked list with the header **modules**. Just as the processes listing, it just needs getting the pointer variable **modules** to get all kernel modules list. Due to the limited spaces and similarities, we do not give the details and only present the experimental results.

2) Experimental Results

In Table 2(we have omit some list for the limited space), we can see that the same kernel modules listing on both Host OS and Guest OS, it verifies that monitoring applications using libvmi library APIs can get the correct result.

Table 2 The Kernel Modules Listing Obtained from Host OS and Guest OS

Host OS	Guest OS
[root@xionghaiquan examples]# ./module-list fedora13 fuse ipt_MASQUERADE iptable_nat nf_nat …… uinput microcode pcspkr	[root@xionghaiquan ~]# lsmod Module Size Used by fuse 50908 2 ipt_MASQUERADE 1716 1 iptable_nat 3961 1 nf_nat 15941 2 …… uinput 5295 0 microcode 10323 0 pcspkr 1394 0

V. RELATED WORK

As bridging the semantic gap is a requirement for all VMI applications and is beneficial for VMM resource management, this challenge has been addressed in several ways: explicit, implicit and new virtualization interface.

The explicit way relies on some specific information, such as the debugging symbols, debugging libraries and Guest OS semantic information to interpret the meaning of variables, structures and functions observed in monitoring applications. The information is generally obtained by reading source code or from binary reverse engineering[3]. In this way, rich, detailed information can be available. However, it is not convenient and easy to maintain because the explicit information changes with Guest OS new version releasing.

The implicit way uses the general operating system algorithms and hardware architectural knowledge to derive the semantic meaning of different events like page faults, hardware interrupts, configuration register updates and etc.[5; 7; 14] This method can reflect the commonalities of

operating systems, so it's more general, applicable and portable across different VMMs and Guest OSes whether they are open source or not.

Another latest one is to create new interfaces for VMM or VMs to obtain the target virtual machine information. It is called symbiotic virtualization[13; 15], a new approach to designing virtualized systems that are capable of fully bridging the semantic gap. Symbiotic virtualization bridges the semantic gap via a bidirectional set of synchronous and asynchronous communication channels. Unlike existing virtualization interfaces, symbiotic virtualization places an equal emphasis on both semantic richness and legacy compatibility. The goal of symbiotic virtualization is to introduce a virtualization interface that provide access to high level semantic information while still retaining the universal compatibility of a virtual hardware interface.

On the whole, the Libvmi's philosophy is similar to the second implicit way.

VI. CONLUSIONS

This paper talks about the issue of semantic gap, a problem results from the lack of Guest OS information within VMM. Through reviewing the architecture of virtual computer systems and the basic interactions between Guest OS and VMM, we show what causes the semantic gap. Then, through illustrating the design, implementation and applications of Libvmi, we can see it can be easily to extend to include other software abstractions just like disk, network, CPU registers and etc. Even further, it can also be extended to provide a VMM resource management optimization experiment platform. For example, with the extensions, we can estimate the dynamic memory demands or identify the workload types in the target Guest OS so that it can help the VMM make better decisions.

ACKNOWLEDGMENT

This work is in part supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302501, the National Science Foundation for Distinguished Young Scholars of China under Grant No. 60925009, the Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant No. 60921002, the Beijing science and technology plans under Grant No.2010B058 and the National Natural Science Foundation of China under Grant No.(61173007，61100013 and 61100015).

REFERENCES

[1] R.P. Goldberg, Survey of Virtual Machine Research. Computer (June 1974) pp.34-45.
[2] B.D.N. Peter M. Chen, When Virtual Is Better Than Real. (2001).
[3] T.G.a.M. Rosenblum., A virtual machine introspection based architecture for intrusion detection. In Proc. Network and Distributed Systems Security Symposium (February 2003.).
[4] J. Pfoh, C. Schneider, and C. Eckert, Exploiting the x86 Architecture to Derive Virtual Machine State Information, Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on, 2010, pp. 166-175.
[5] S.T. Jones, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, Antfarm: tracking processes in a virtual machine environment, Proceedings of the annual conference on USENIX '06 Annual Technical Conference, USENIX Association, Boston, MA, 2006, pp. 1-1.
[6] W. Zhao, and Z. Wang, Dynamic memory balancing for virtual machines, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, Washington, DC, USA, 2009, pp. 21-30.
[7] S.T. Jones, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, Geiger: monitoring the buffer cache in a virtual machine environment. SIGPLAN Not. 41 (2006) 14-24.
[8] G.J. Popek, and R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Proceedings of the fourth ACM symposium on Operating system principles, ACM, 1973, pp. 121.
[9] J.S. Robin, and C.E. Irvine, Analysis of the Intel Pentium's ability to support a secure virtual machine monitor, Proceedings of the 9th conference on USENIX Security Symposium - Volume 9, USENIX Association, Denver, Colorado, 2000, pp. 10-10.
[10] K. Karadeniz, Analysis of Intel IA-64 Processor Support for a Secure Virtual Machine Monitor. (March 2001) 91.
[11] Libvmi, http://code.google.com/p/vmitools/.
[12] M.E.R.a.D.A. Solomon, Microsoft Windows Internals, 5th. Microsoft Press (2009).
[13] J.R. Lange, Symbiotic Virtualization. (September 13, 2010) 226.
[14] H. Kim, H. Lim, J. Jeong, H. Jo, J. Lee, and S. Maeng, Transparently bridging semantic gap in CPU management for virtualized environments. J. Parallel Distrib. Comput. 71 (2011) 758-773.
[15] J.R. Lange, and P. Dinda, SymCall: symbiotic virtualization through VMM-to-guest upcalls, Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, Newport Beach, California, USA, 2011, pp. 193-204.