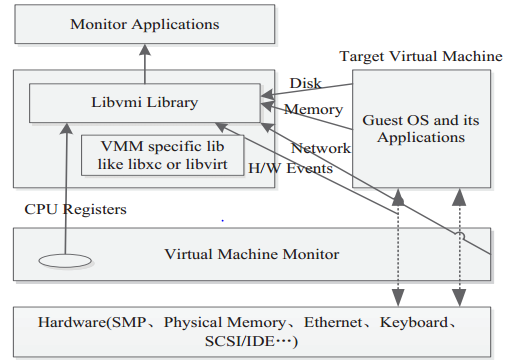
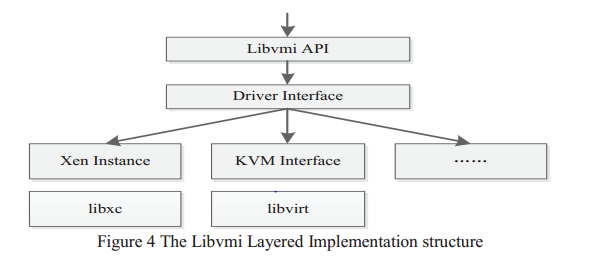
**关于LIBVMI**

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



The Overall Libvmi Architecture

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 isequivalent 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.

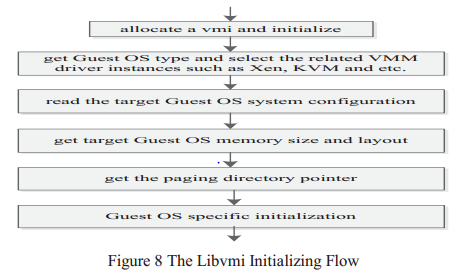


On the whole, the Libvmi APIs can be divided into two categories: one is the library initializing and destroying management APIs, the other is utility APIs for different kinds of purposes, such as memory accesses APIs, disk accesses APIs, CPU registers accesses APIs and etc.

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.



typedef struct xen\_instance {

libvmi\_xenctrl\_handle\_t xchandle; /\*\*< handle to xenctrl library (libxc) \*/

unsigned long domainid; /\*\*< domid that we are accessing \*/

int xen\_version; /\*\*< version of Xen libxa is running on \*/

int hvm; /\*\*< nonzero if HVM \*/

xc\_dominfo\_t info; /\*\*< libxc info: domid, ssidref, stats, etc \*/

uint8\_t addr\_width; /\*\*< guest's address width in bytes: 4 or 8 \*/

#ifdef HAVE\_LIBXENSTORE

struct xs\_handle \*xshandle; /\*\*< handle to xenstore daemon \*/

#endif

char \*name;

#if ENABLE\_XEN\_EVENTS==1

xen\_events\_t \*events; /\*\*< handle to events data \*/

#endif

}

* vmi\_init(&vmi, **VMI\_AUTO | VMI\_INIT\_COMPLETE**, name)
* vmi\_init\_private(vmi, **flags** | VMI\_CONFIG\_GLOBAL\_FILE\_ENTRY, VMI\_INVALID\_DOMID, name, NULL);
  + set\_driver\_type🡪 xen\_test🡪 xen\_get\_domainid\_from\_name🡪 xs\_read逐一读取虚拟机名字与给定name相比较vmi->mode = VMI\_XEN;
  + set\_id\_and\_name /\* resolve the id and name \*/
  + driver\_init /\* driver-specific initilization \*/ 🡪xen\_init
    - driver\_get\_instance
      * driver\_xen\_setup(vmi);

instance->init\_ptr = &xen\_init;

instance->destroy\_ptr = &xen\_destroy;

…………………………………………………………..

/\* read and parse the config file 读取虚拟机相应配置文件 \*/

* read\_config\_string
* read\_config\_file\_entry
* set\_os\_type\_from\_config
* init\_page\_offset
* driver\_get\_memsize
* get\_memory\_layout
* /\* setup OS specific stuff \*/
  + linux\_init
    - os\_interface->os\_get\_offset = linux\_get\_offset;
    - os\_interface->os\_pid\_to\_pgd = linux\_pid\_to\_pgd;

…………………………………………………………………………..

* windows\_init
  + os\_interface->os\_get\_offset = windows\_get\_offset;
  + os\_interface->os\_pid\_to\_pgd = windows\_pid\_to\_pgd;

………………………………………………………………………………….

b) Libvmi Utility API Examples: 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.

**分析：vmi\_read\_addr\_ksym首先把内核符号转化为虚拟地址，vmi\_read\_addr\_va首先把虚拟地址转化为相应的物理地址，所以最终都要以vmi\_read\_addr\_pa形式读取。**

**vmi\_read\_addr\_pa🡪vmi\_read\_64\_pa🡪vmi\_read\_X\_pa🡪vmi\_read\_pa🡪vmi\_read\_page由初始化可知即xen\_read\_page(instance->read\_page\_ptr = &xen\_read\_page;)**

**xen\_read\_page🡪memory\_cache\_insert🡪validate\_and\_return\_data🡪get\_memory\_data🡪get\_data\_callback**

**在xen\_init初始化中: memory\_cache\_init(vmi, xen\_get\_memory, xen\_release\_memory, 0);查看memory\_cache\_init定义可知：**

**void memory\_cache\_init (**

**vmi\_instance\_t vmi,**

**void \*(\*get\_data)(vmi\_instance\_t, addr\_t, uint32\_t),**

**void (\*release\_data)(void \*, size\_t),**

**unsigned long age\_limit)**

**{**

**………………………………………………………………………………………..**

**vmi->memory\_cache\_size\_max = MAX\_PAGE\_CACHE\_SIZE;**

**get\_data\_callback = get\_data;**

**release\_data\_callback = release\_data;**

**}**

**利用以上映射关系调用get\_data\_callback即实现调用xen\_get\_memory**

**xen\_get\_memory🡪xen\_get\_memory\_pfn🡪xc\_map\_foreign\_range🡪 linux\_privcmd\_map\_foreign\_range🡪xc\_map\_foreign\_pages🡪xc\_map\_foreign\_bulk🡪 linux\_privcmd\_map\_foreign\_bulk**

**注：vmi\_translate\_kv2p /\* expose virtual to physical mapping for kernel space via api call \*/**

**vmi\_translate\_uv2p/\* expose virtual to physical mapping for user space via api call \*/**

**都是先确定cr3，然后程序模拟遍历虚拟机页表得到相应虚拟机物理地址。**



如**vmi\_translate\_kv2p🡪** v2p\_nopae

走第一层，得到pgd\_entry(cr3指向页初始地址，最后12位为零；第一层页表项10位，即每页共1024个页表项，每个页表项占32位)

pgd = get\_pgd\_nopae(vmi, vaddr, dtb);

dbprint("--PTLookup: pgd = 0x%.8"PRIx32"\n", pgd); 可理解如下：

***uint32\_t pgd\_entry = (pdpe & 0xFFFFF000) + ((((address) >> 22) & 0x3FF) \* sizeof(uint32\_t));***

***size\_t len\_read = vmi\_read\_pa(vmi, pgd\_entry, &value, 4);***

类似地，走第二层，得到pte\_entry:

pte = get\_pte\_nopae(vmi, vaddr, pgd);

dbprint("--PTLookup: pte = 0x%.8"PRIx32"\n", pte);

最后，利用pte得到相应的虚拟机物理地址：

paddr = get\_paddr\_nopae(vaddr, pte);