An Introduction to Virtual Machine Introspection Using LibVMI

Bryan D. Payne
bdpayne@acm.org

nebula®
(cloud) Computing for the Enterprise

Virtual Machine Introspection

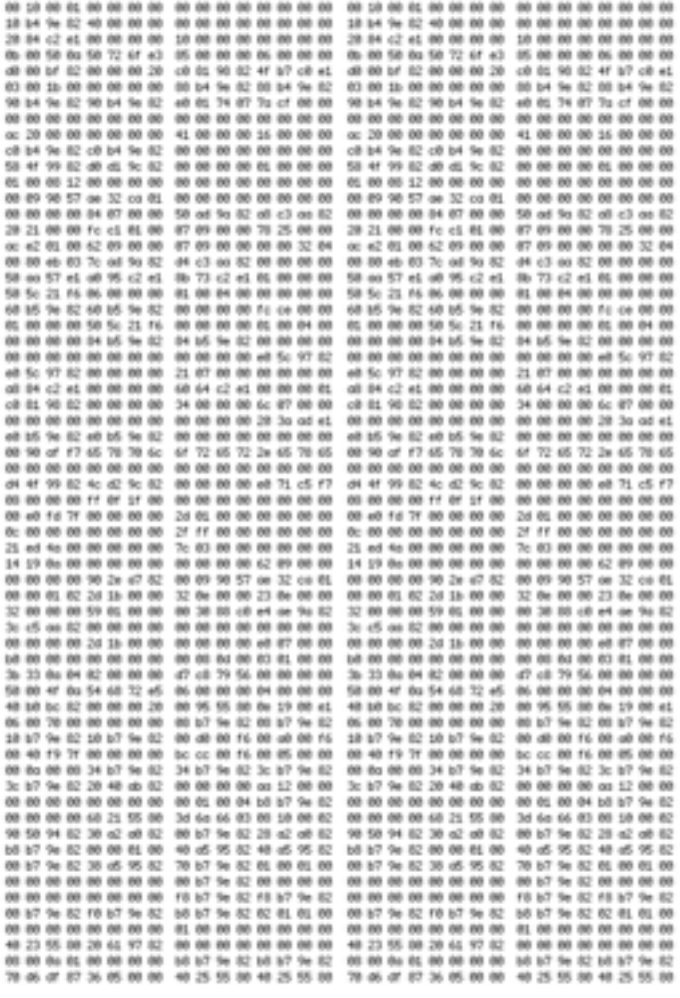
* Memory Analysis (at runtime)
* Events
  - Register changes
  - Memory read / write / execute
  - Memory mapped I/O
  - Exceptions (e.g., page faults)

Use Cases

* Forensics
* System-level debugging and analysis
* Runtime security
* Timeline or trend analysis
* Debugging
* Other ideas?

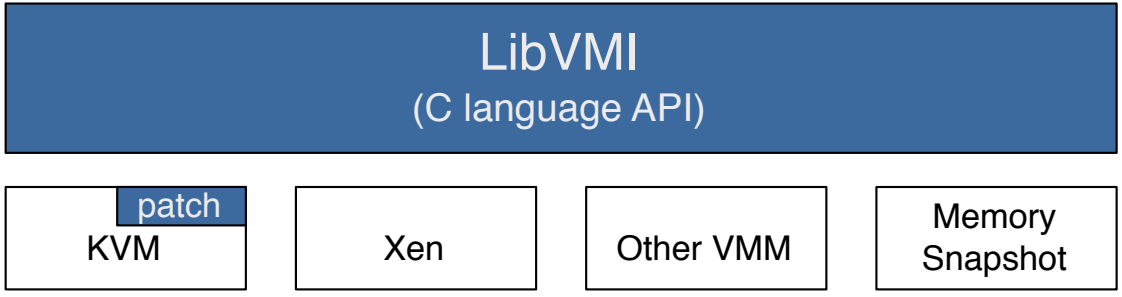
Hard Problems

* Semantic Gap
* Performance
* Platform Support

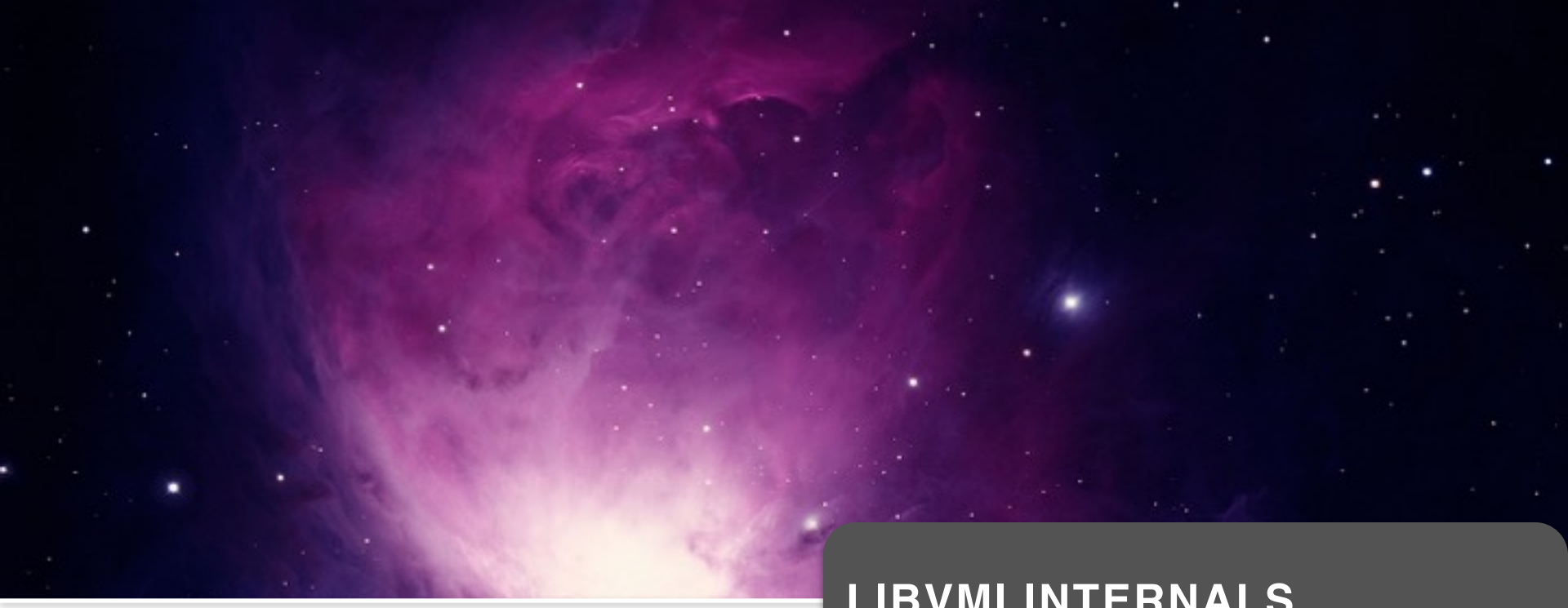


nebula

LibVMI



* Goal: Make VMI more accessible to programmers
  - Backends: KVM/QEMU, Xen, Raw snapshot files
  - Operating Systems: Linux, Windows
  - Architectures: x86 (32-bit, PAE, 64-bit), ARM
  - API: read/write memory, memory events, translations, cache management



LIBVMI INTERNALS

nebula

LibVMI Internals

* Initialization of LibVMI
* Runtime guest introspection
  - Memory access (read / write)
  - Register access
  - Memory events
  - Address translation
  - Symbol resolution
  - Cache management
* Hypervisor-level support

LibVMI Initialization

```
vmi_init(…)
vmi_init_complete(…)
```

- Find VM (Xen, KVM, etc)
- Read config file
- Memory layout and size
- Find kernel base location
- Init symbol resolution (exports, debug info, etc)
- Find page directory location (CR3 / kpgd)
- Find kernel process list

LibVMI Initialization

```
vmi_init(…)
vmi_init_complete(…)
```

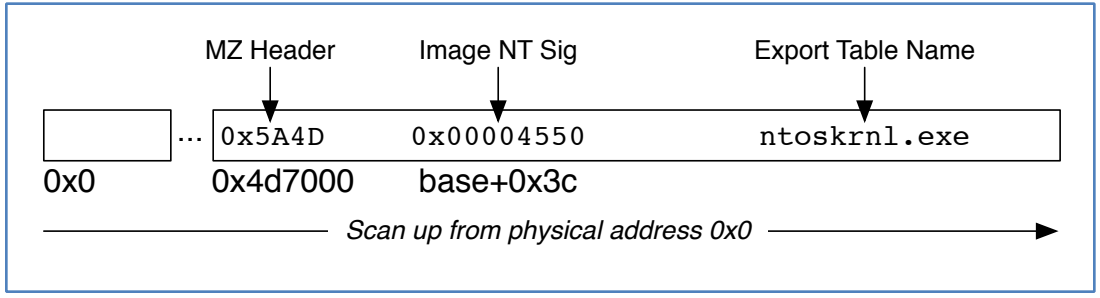
- ~~Find VM (Xen, KVM, etc)~~
- ~~Read config file~~
- ~~Memory layout and size~~
- Find kernel base location
- Init symbol resolution (exports, debug info, etc)
- Find page directory location (CR3 / kpgd)
- Find kernel process list

Config file: libvmi.conf

```
winxpsp2 {
    ostype = "Windows";
    win_tasks   = 0x88;
    win_pdbase  = 0x18;
    win_pid     = 0x84;
    win_kdvb    = 0x80544ce0;
}

win7sp1x64 {
    ostype = "Windows";
    win_tasks   = 0x188;
    win_pdbase  = 0x28;
    win_pid     = 0x180;
    win_kdvb    = 0xfffff800027f10a0;
}
```

Finding the Kernel

1.

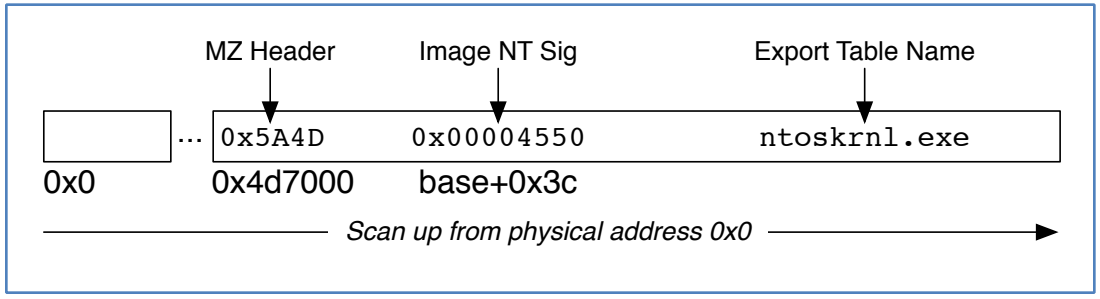


2.

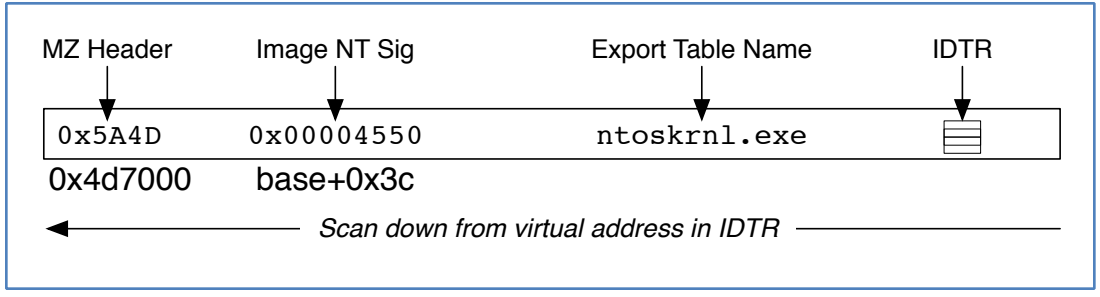
3.

Finding the Kernel

1.



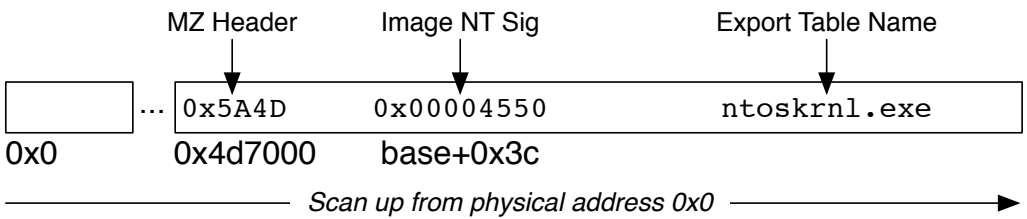
2.



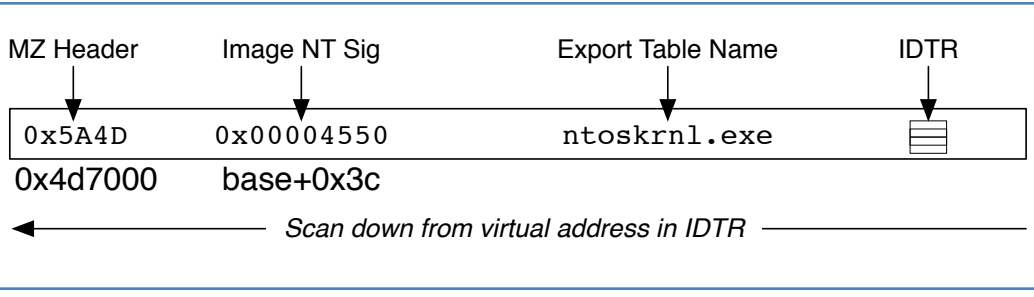
3.

Finding the Kernel

1.

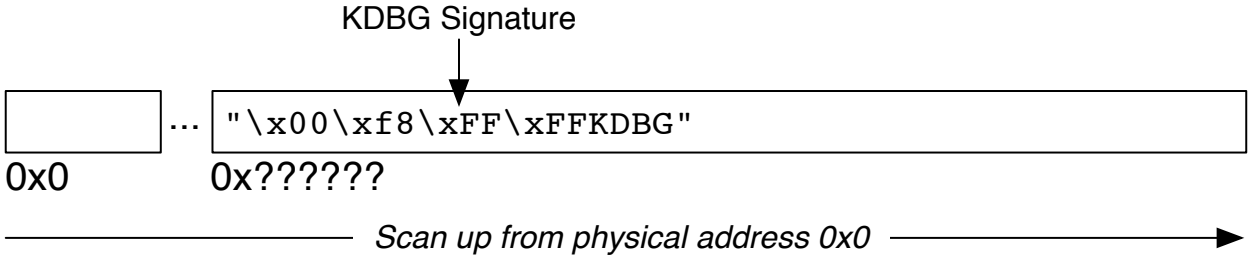


2.



3. _KDDEBUGGER_DATA64[KernBase]

Where Is _KDDEBUGGER_DATA64?



- May take longer than just finding kernel directly
- Symbol access makes it all worthwhile
  - KernBase
  - PsLoadedModuleList
  - 125+ symbols

LibVMI Initialization

```
vmi_init(…)
vmi_init_complete(…)
```

- ~~Find VM (Xen, KVM, etc)~~
- ~~Read config file~~
- ~~Memory layout and size~~
- ~~Find kernel base location~~
- Init symbol resolution (exports, debug info, etc)
- Find page directory location (CR3 / kpgd)
- Find kernel process list

Symbol Resolution

* _KDDEBUGGER_DATA64
* Kernel PE Export Table
  - Start with kernel base location
  - Parse kernel PE header
  - RVA to export table in optional head data dir

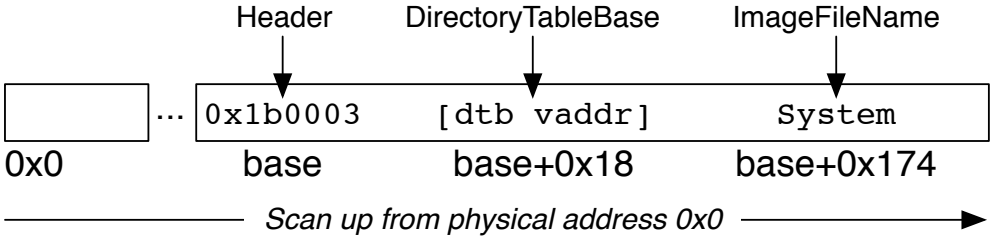
LibVMI Initialization

```
vmi_init(…)
vmi_init_complete(…)
```

- ~~Find VM (Xen, KVM, etc)~~
- ~~Read config file~~
- ~~Memory layout and size~~
- ~~Find kernel base location~~
- ~~Init symbol resolution (exports, debug info, etc)~~
- Find page directory location (CR3 / kpgd)
- Find kernel process list

Page Directory

* CR3, or Search for “System” EPROCESS struct




```
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
      +0x000 Header           : _DISPATCHER_HEADER
      +0x010 ProfileListHead  : _LIST_ENTRY
      +0x018 DirectoryTableBase : [2] Uint4B
      +0x020 LdtDescriptor    : _KGDTENTRY
      +0x028 Int21Descriptor  : _KIDTENTRY
      +0x030 IopmOffset       : Uint2B
      ...
      +0x000 Count            : Uint4B
      +0x000 Ptr              : Ptr32 Void
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
      +0x000 Flink            : Ptr32 _LIST_ENTRY
      +0x004 Blink            : Ptr32 _LIST_ENTRY
    ...
```


nebula

LibVMI Initialization

```
vmi_init(…)
vmi_init_complete(…)
```

- ~~Find VM (Xen, KVM, etc)~~
- ~~Read config file~~
- ~~Memory layout and size~~
- ~~Find kernel base location~~
- ~~Init symbol resolution (exports, debug info, etc)~~
- ~~Find page directory location (CR3 / kpgd)~~
- Find kernel process list

LibVMI Initialization

```
vmi_init(…)
vmi_init_complete(…)
```

- ~~Find VM (Xen, KVM, etc)~~
- ~~Read config file~~
- ~~Memory layout and size~~
- ~~Find kernel base location~~
- ~~Init symbol resolution (exports, debug info, etc)~~
- ~~Find page directory location (CR3 / kpgd)~~
- ~~Find kernel process list~~

Using symbol from _KDDEBUGGER_DATA64 (PsActiveProcessHead)

LibVMI Initialization

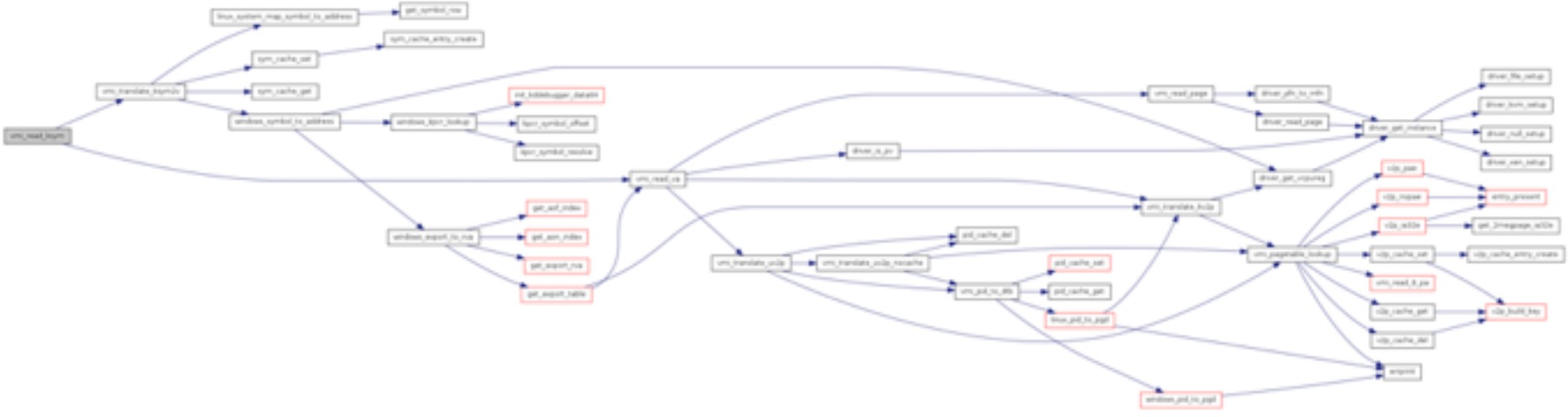
```
vmi_init(…)
vmi_init_complete(…)
```

- ~~Find VM (Xen, KVM, etc)~~
- ~~Read config file~~
- ~~Memory layout and size~~
- ~~Find kernel base location~~
- ~~Init symbol resolution (exports, debug info, etc)~~
- ~~Find page directory location (CR3 / kpgd)~~
- ~~Find kernel process list~~

LibVMI Runtime

* Read / Write Functions
  - Starting from Kernel Symbol, Vaddr, or Paddr
  - Specify length to read
  - Read a string (ASCII or UNICODE)
* Address Translation Functions
  - Kernel or User Vaddr to Paddr
  - Kernel symbol to Vaddr
* Convenience Functions
  - Pause / Resume, Memory size, CPU Registers
  - LibVMI cache manipulation

Read Example (vmi_read_ksym)



resolve symbol

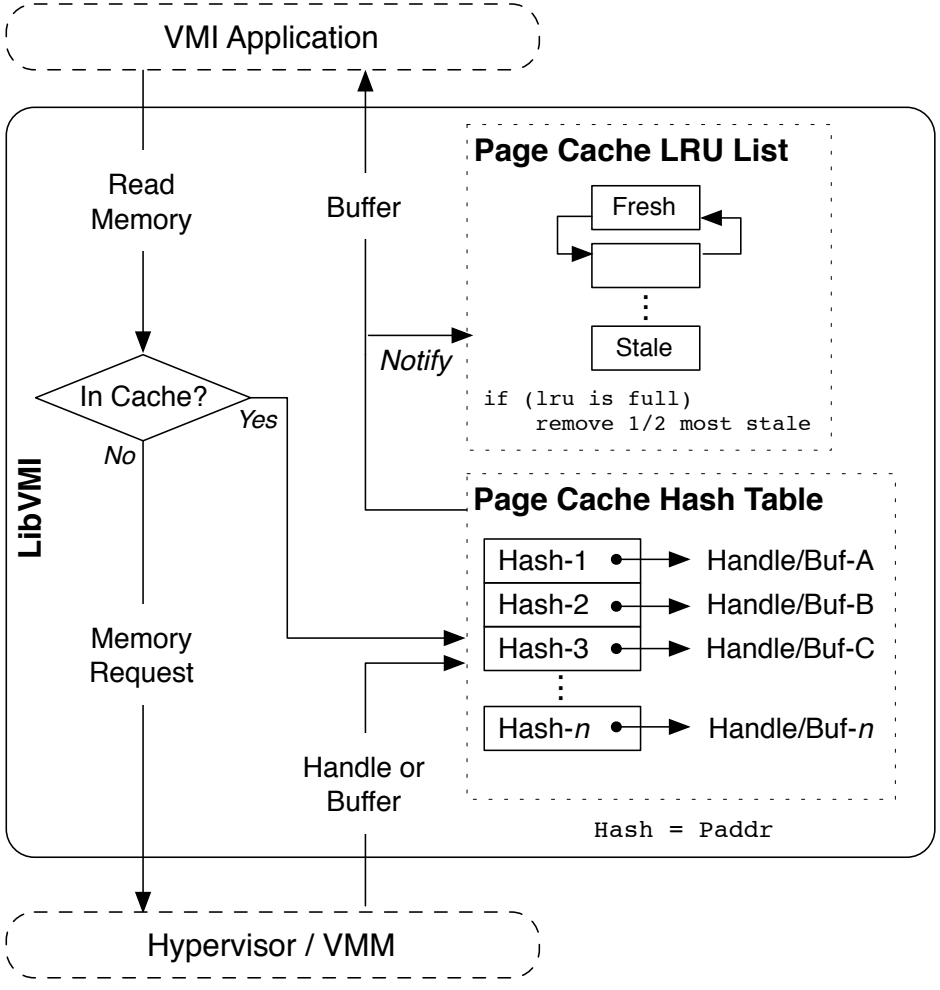
translate to paddr

read from VMM

handle page wraps

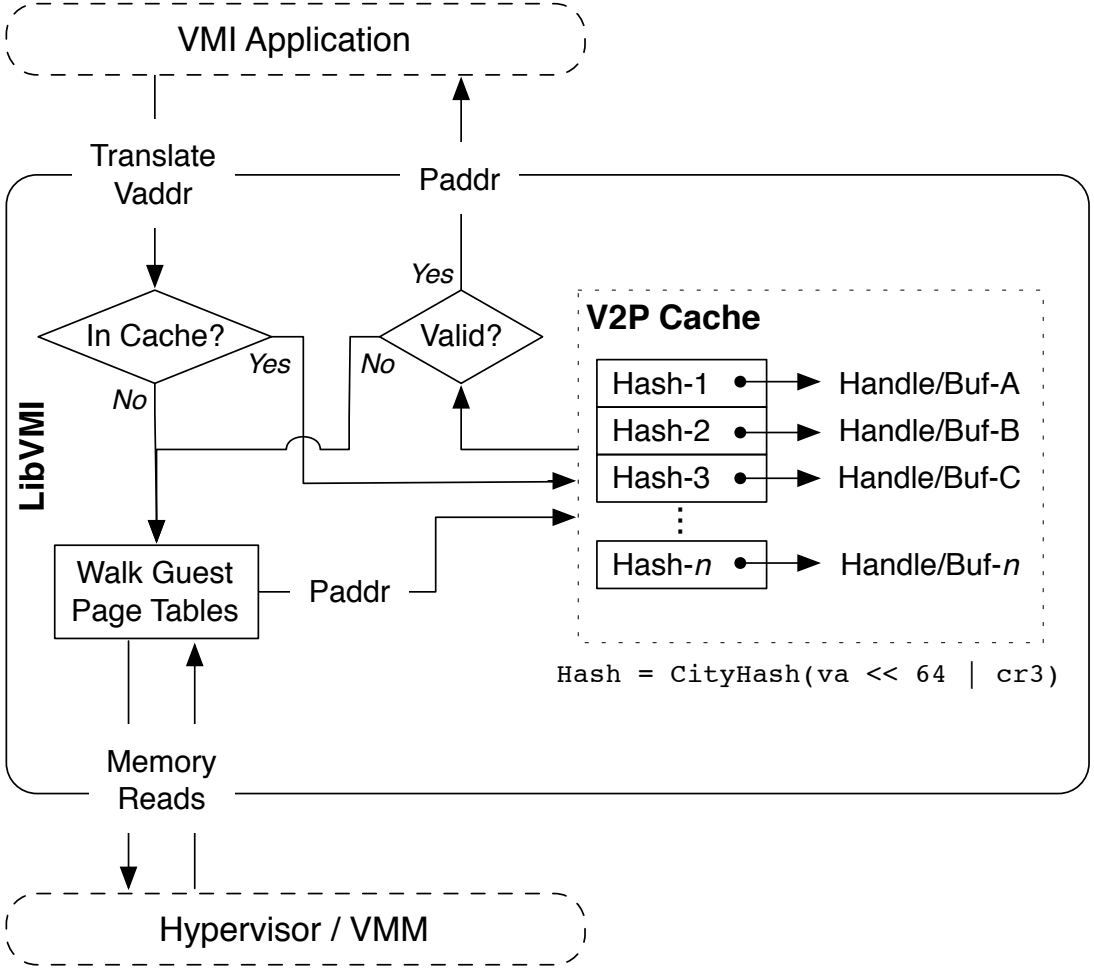
nebula

Page-level Cache



nebula

Virtual To Physical Cache



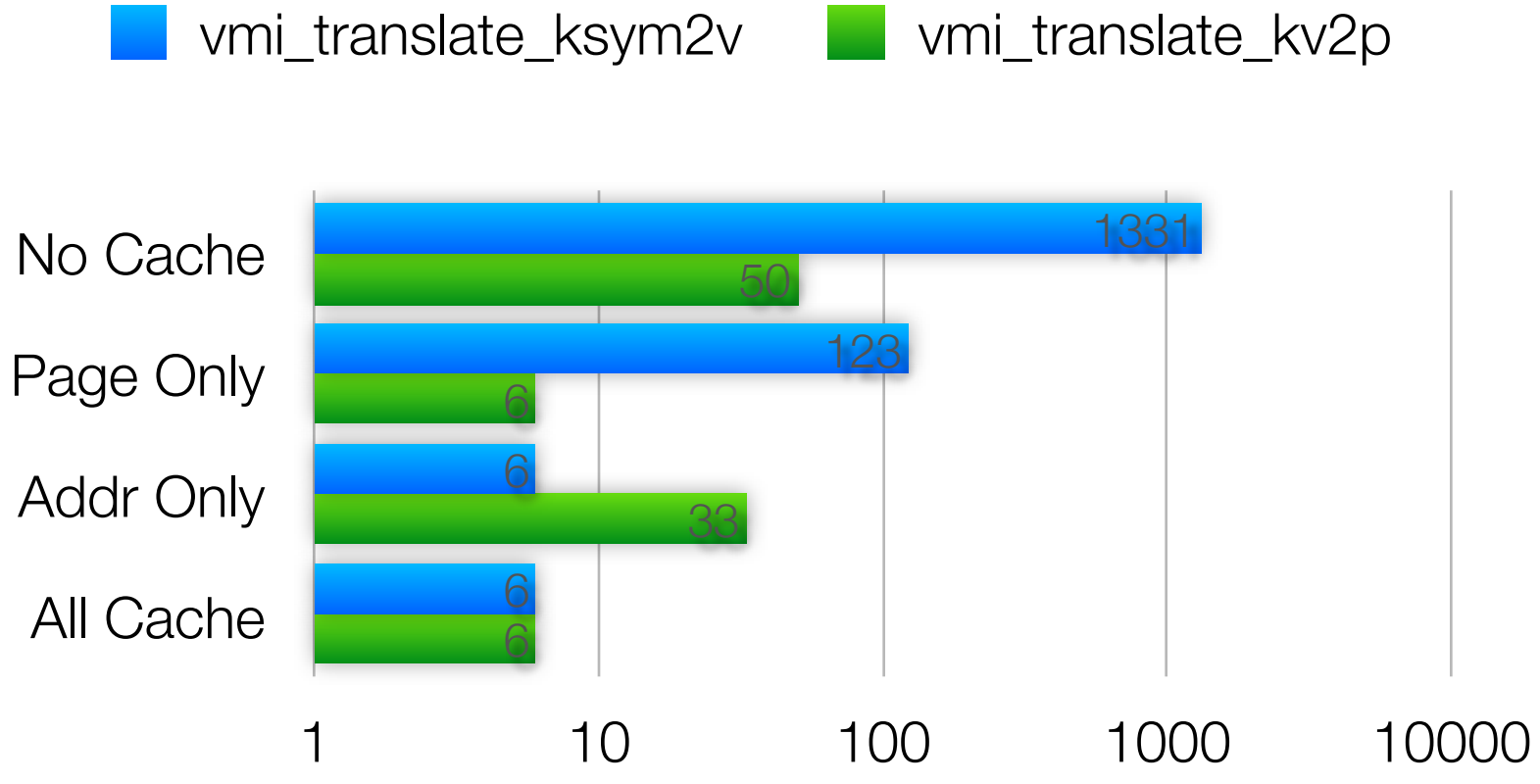
nebula

Cache Summary

- Page-level data
- Virtual address to Physical address
- Process ID to Directory Table Base
- Kernel Symbol to Virtual address

Cache Performance

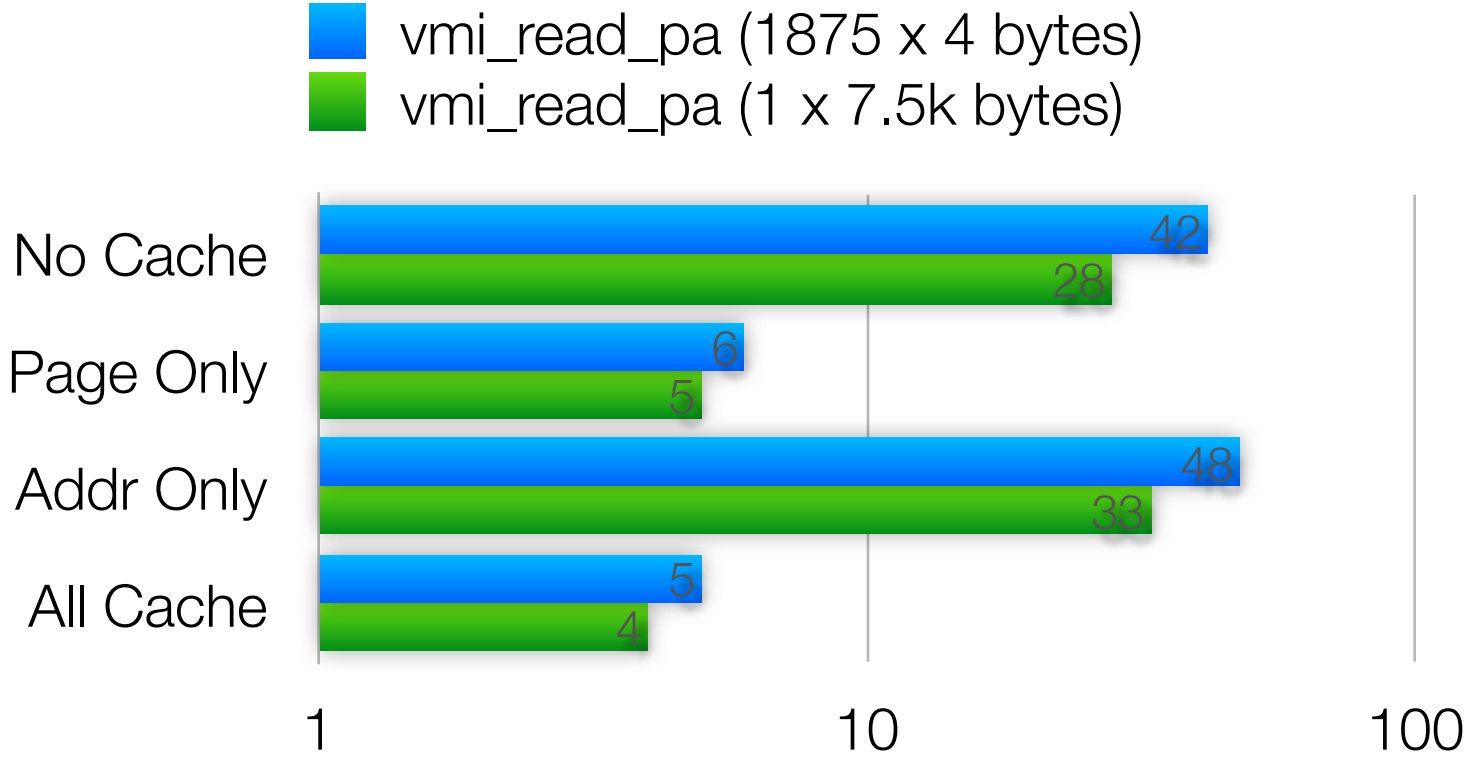
System configuration: Xen 4.1.1, Dual Intel Xeon X5675, 24G RAM, Windows XP VM
Times shown are for cache hits, when possible



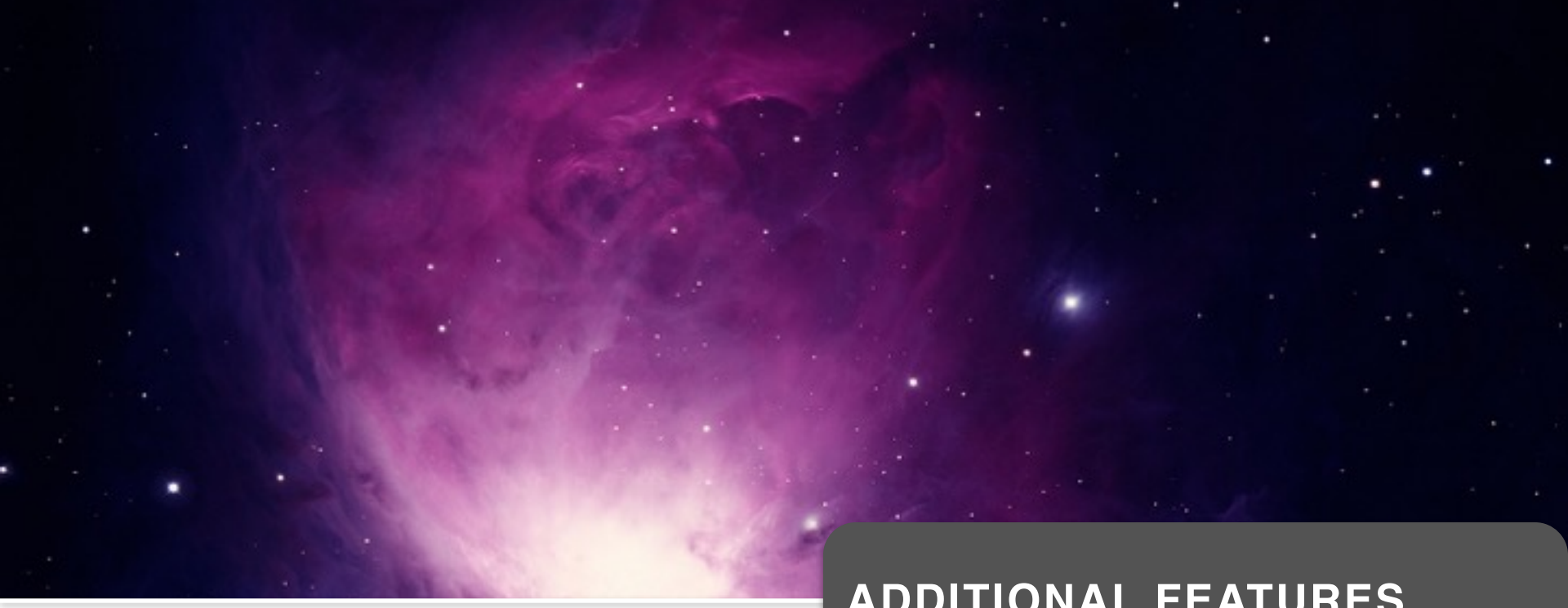
nebula

Cache Performance

System configuration: Xen 4.1.1, Dual Intel Xeon X5675, 24G RAM, Windows XP VM
Times shown are for cache hits, when possible



nebula



ADDITIONAL FEATURES

Events (Xen)

- Pause guest and transfer control to callback function in your application
- Memory r/w/x events on defined regions
- Register r/w events on CR0/CR3/CR4/MSR regs
- Interrupt events
- Single step through instructions

Shared Memory Snapshots (KVM)

- Requires custom patch for Qemu-KVM
- Transparently creates a guest snapshot
- Guest continues running
- VMI app gets direct memory access
- VMI app can refresh snapshot at will

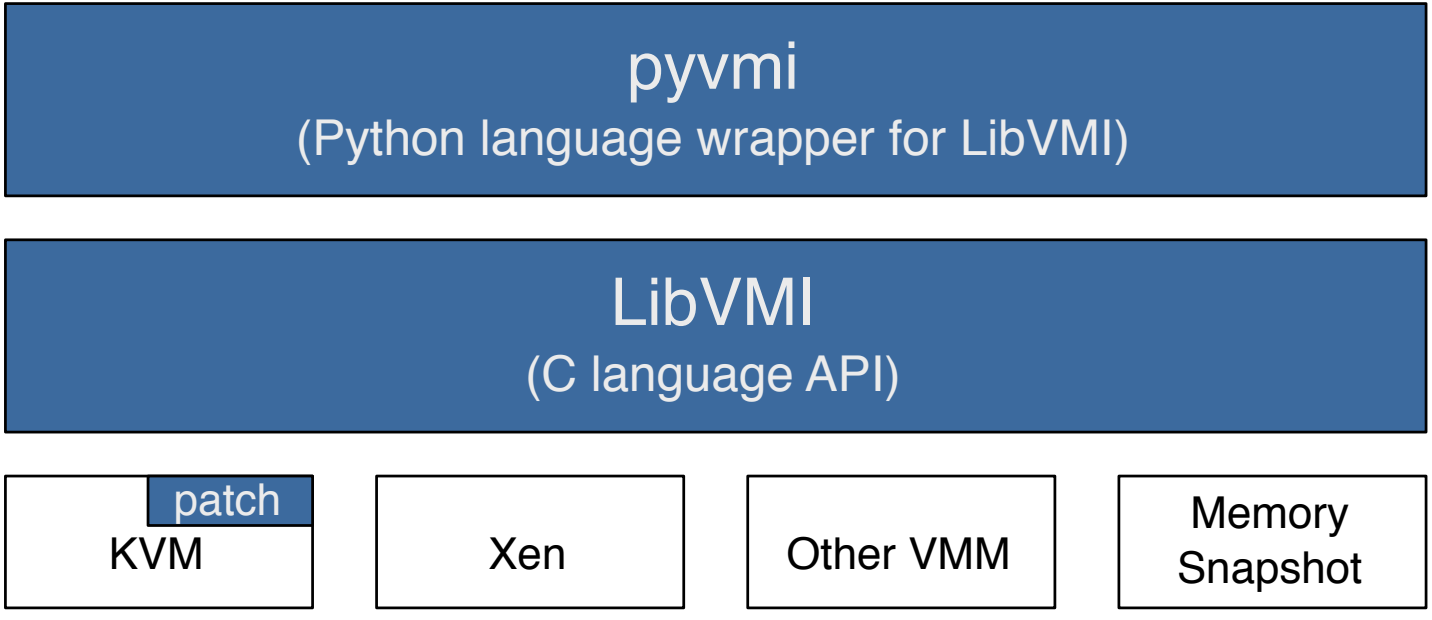
Rekall Profiles (Windows)

- Use Rekall tool to generate Windows profiles
- Profiles replace the need to provide offsets in the libvmi.conf file
- Especially useful for Windows 8, where KDBG is typically not accessible

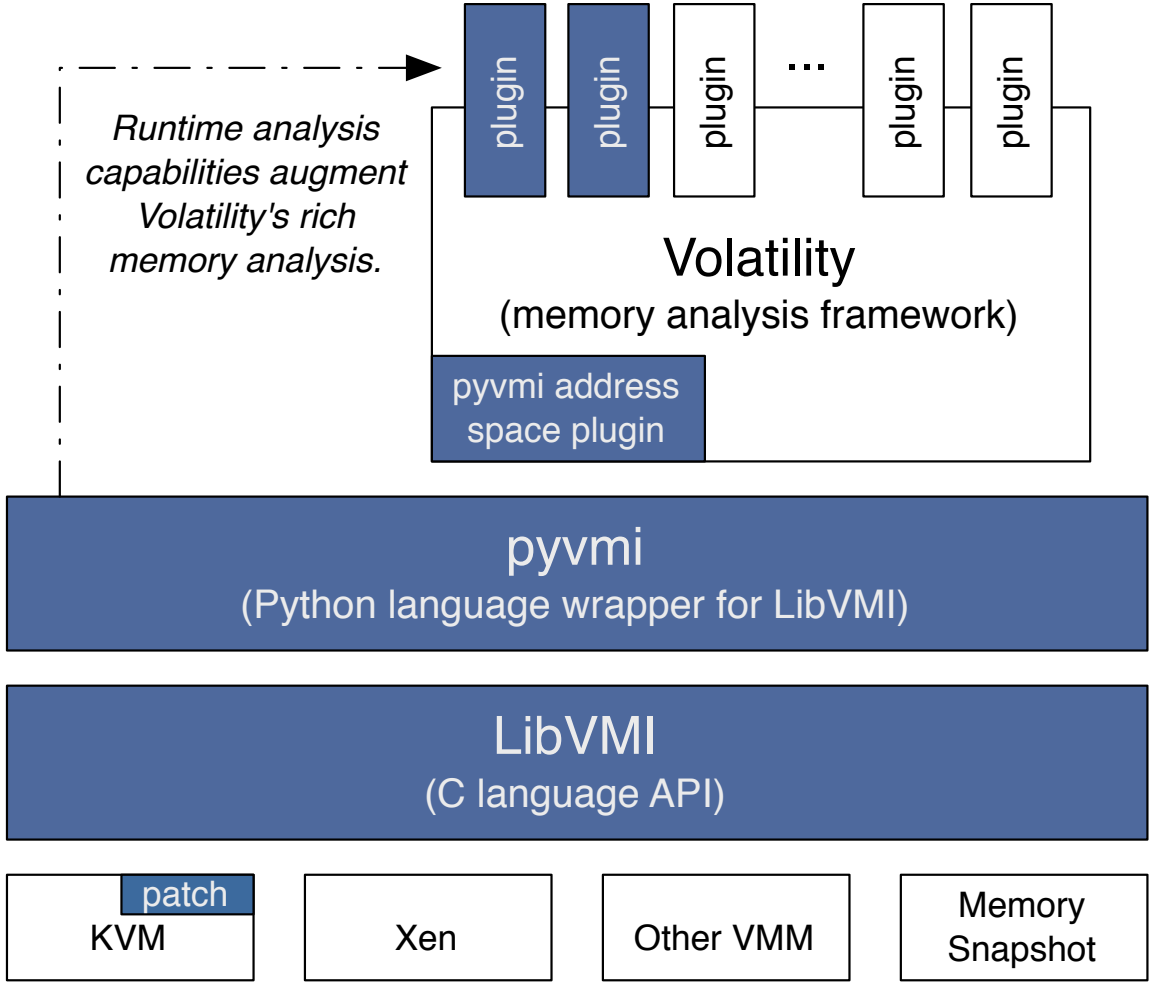
USING LIBVMI

nebula

```c
int main (int argc, char **argv)
{
    vmi_instance_t vmi;
    addr_t next_process, list_head;
    char *procname = NULL;
    int pid = 0;
    int tasks_offset, pid_offset, name_offset;

    vmi_init(&vmi, VMI_AUTO | VMI_INIT_COMPLETE, argv[1]);
    tasks_offset = vmi_get_offset(vmi, "win_tasks");
    name_offset = vmi_get_offset(vmi, "win_pname");
    pid_offset = vmi_get_offset(vmi, "win_pid");

    vmi_read_addr_ksym(vmi, "PsInitialSystemProcess", &list_head);
    vmi_read_addr_va(vmi, list_head + tasks_offset, 0, &next_process);
    vmi_read_32_va(vmi, list_head + pid_offset, 0, &pid);
    vmi_read_32_va(vmi, list_head + pid_offset, 0, &pid);
    procname = vmi_read_str_va(vmi, list_head + name_offset, 0);
    printf("[   ]     ", pid, procname);
    if (procname){
        free(procname);
        procname = NULL;
    }
    list_head = next_process;

    while (1){
        addr_t tmp_next = 0;
        vmi_read_addr_va(vmi, next_process, 0, &tmp_next);
        if (list_head == tmp_next){
            break;
        }

        procname = vmi_read_str_va(vmi, next_process + name_offset - tasks_offset, 0);
        vmi_read_32_va(vmi, next_process + pid_offset - tasks_offset, 0, &pid);

        if (pid >= 0 && procname){
            printf("[   ]     ", pid, procname);
        }
        if (procname){
            free(procname);
            procname = NULL;
        }
        next_process = tmp_next;
    }

    vmi_destroy(vmi);
    return 0;
```



```
import sys
from pyvmi.pyvmi import Pyvmi


def get_processes(vmi):
    """ Generator returns address of each task_struct / EPROCESS struct."""
    if vmi['ostype'] == 'Linux':
        current_process = vmi.translate(ksym='init_task')
    elif vmi['ostype'] == 'Windows':
        current_process = vmi.read(ksym='PsInitialSystemProcess')

    list_head = current_process + vmi['tasks_offset']
    next_list_entry = vmi.read(va=list_head)

    while (next_list_entry != list_head):
        yield(current_process)
        current_process = next_list_entry - vmi['tasks_offset']
        next_list_entry = vmi.read(va=next_list_entry)


def get_pid_and_proc(vmi):
    """ Generator returns pid, process name tuples for each running process."""
    process_structs = get_processes(vmi)
    for struct in process_structs:
        procname = vmi.read(va=struct + vmi['name_offset'], string=True)
        pid = vmi.read(va=struct + vmi['pid_offset'], size=4)
        if (pid < (1 << 16)):
            yield pid, procname


def main(argv):
    """ Prints process information for given target."""
    with Pyvmi(argv[1]) as vmi:
        for pid, procname in get_pid_and_proc(vmi):
            print "[%5d] %s" % (pid, procname)
```



Development

https://github.com/libvmi/libvmi

Discussion

https://groups.google.com/d/forum/vmitools

An Introduction to Virtual Machine Introspection Using LibVMI

Bryan D. Payne
bdpayne@acm.org

nebula®
(cloud) Computing for the Enterprise