eKimono: Detecting Rootkits inside Virtual Machines

DeepSec 2009, 19th-20th/11/2009

Nguyen Anh Quynh, Kuniyasu Suzaki, Ruo Ando

The National institute of Advanced Industrial Science & Technology (AIST), Japan

Who am I?

- NGUYEN Anh **Quynh**, a researcher working in Japan.
  - National Institute of Advanced Industrial Science & Technology (AIST), Japan
  - PhD degree in Computer Science from Keio University, Japan
  - A member of Vnsecurity.net
  - Interests: Operating System, Virtualization, Trusted computing, IDS, malware, digital forensic, ...

Agenda

- Problems of current malware scanner
- eKimono: Malware detector for Virtual Machines
  - Introduction on Virtual machine
  - Architecture, design and implementation of eKimono
    - Focus on Windows protection
    - Focus more on rootkit detection in this talk
- eKimono demo on detecting malware
- Conclusions

Part I

- **Problems of current malware scanner**
  - Focus on rootkits
- eKimono: Malware detector for Virtual Machines
  - Introduction on Virtual machine
  - Architecture, design and implementation of eKimono
    - Focus on Windows VM protection
- eKimono demo on detecting malware
- Conclusions

What is Rootkit?

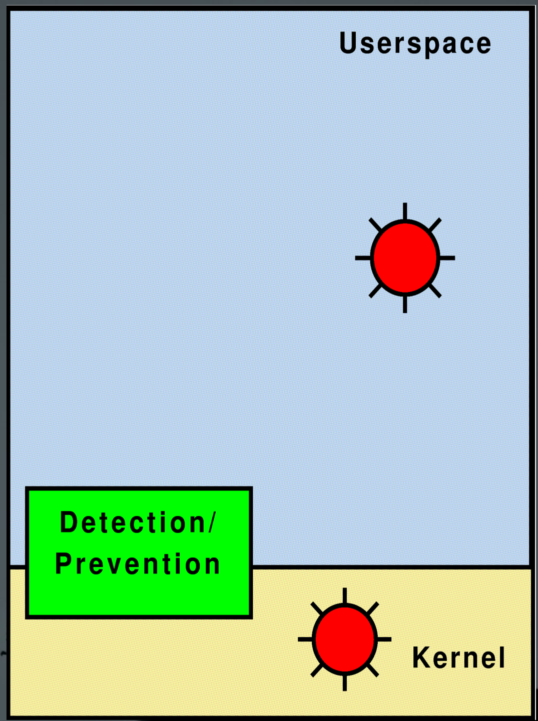
- Malware trying to hide their existence in the system
  - Modify the system tools
    - Trojan system binaries to return faked information
  - Modify system to hook critical functions that can disclose their residence
    - Patch system process at runtime
      - IAT, EAT, Inline hooking
    - Modify system kernel
      - System calls
      - IDT, GDT
      - IAT/EAT
      - Modify kernel objects
        - DKOM technique

Current Malware Scanner

- Run inside the system to scan malware
- Mostly only scan HDD to detect malware

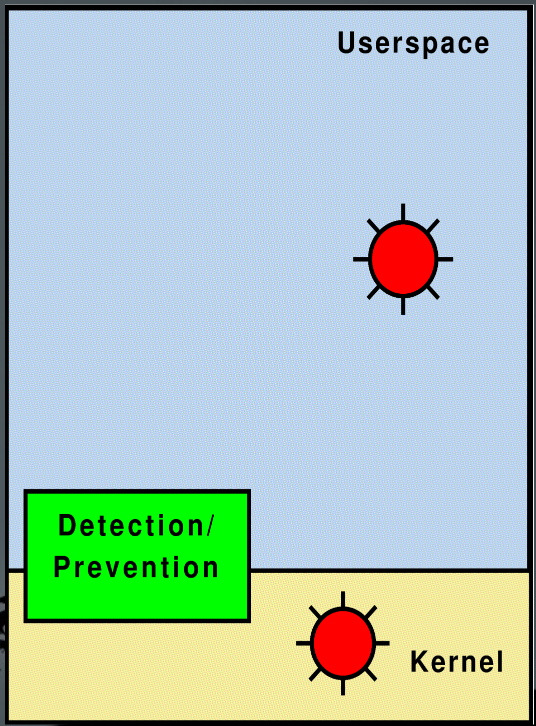
Malware Scanner Problems

- Can be easily fooled by rootkits
  - Return faked information
- Can be easily tampered by rootkits
  - Even being a target of attack



Other Problems ...

- Focus more on scanning HDD, but mostly ignore memory
  - Easily defeated by rootkits/malware that only stay in memory, but never write down itself to HDD!



I Dream a Dream ...

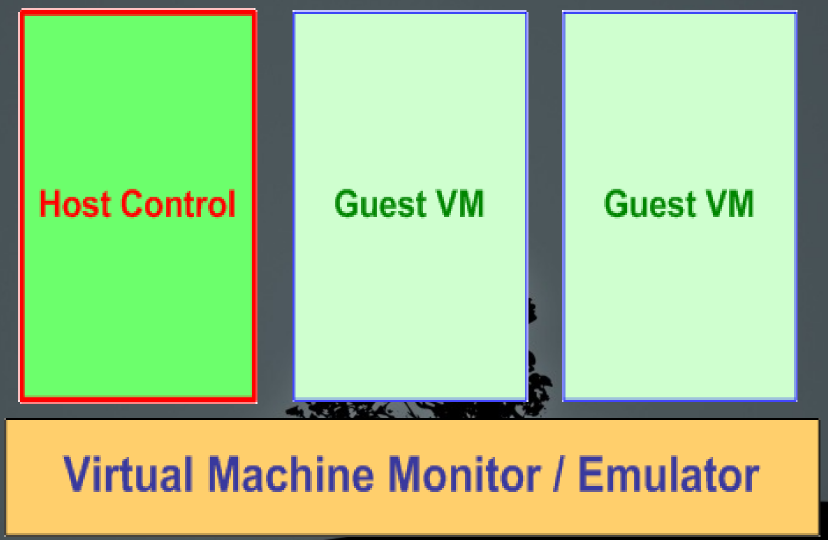
- A perfect malware scanner?
  - Detect malware in memory
  - Not easily be fooled by malware
  - Cannot be, (or very hard to be), tampered by malware
    - Even if malware run in the kernel

Part II

- Problems of current malware scanner
- **eKimono: Rootkit scanner for Virtual Machine**
  - **Introduction on Virtual machine**
  - **Architecture, design and implementation of eKimono**
  - **Focus on Windows protection**
- eKimono demo on detecting malware
- Conclusions

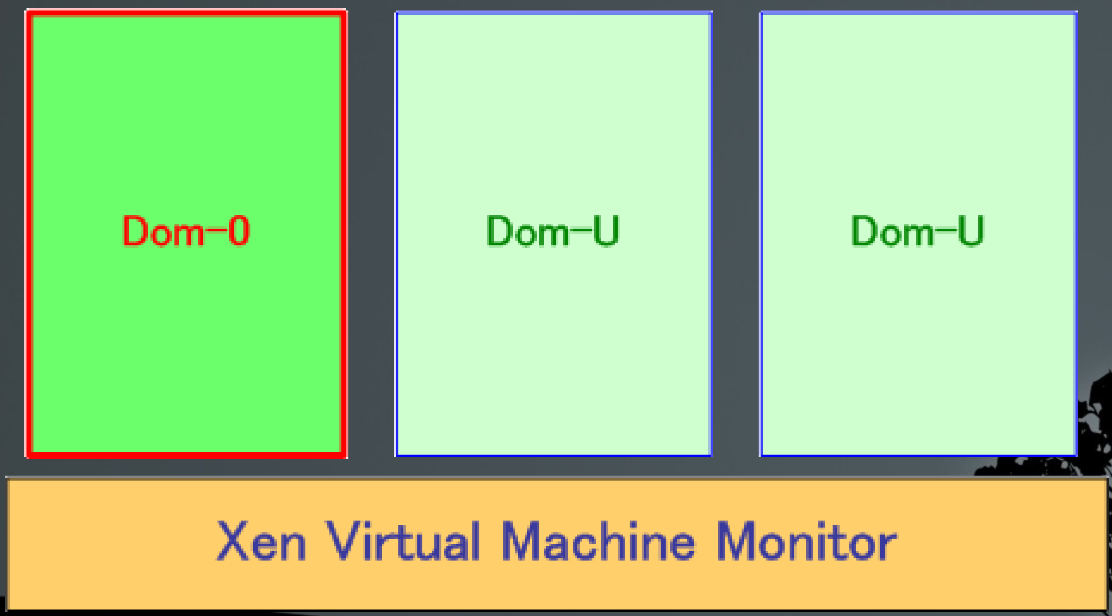
Virtual Machine Concept

- Running multiple virtual systems on a physical machine at the same time
  - Privilege VM
  - Guest VM
  - Multiple Operating Systems are supported
    - Windows, Linux, BSD, MacOSX, ...



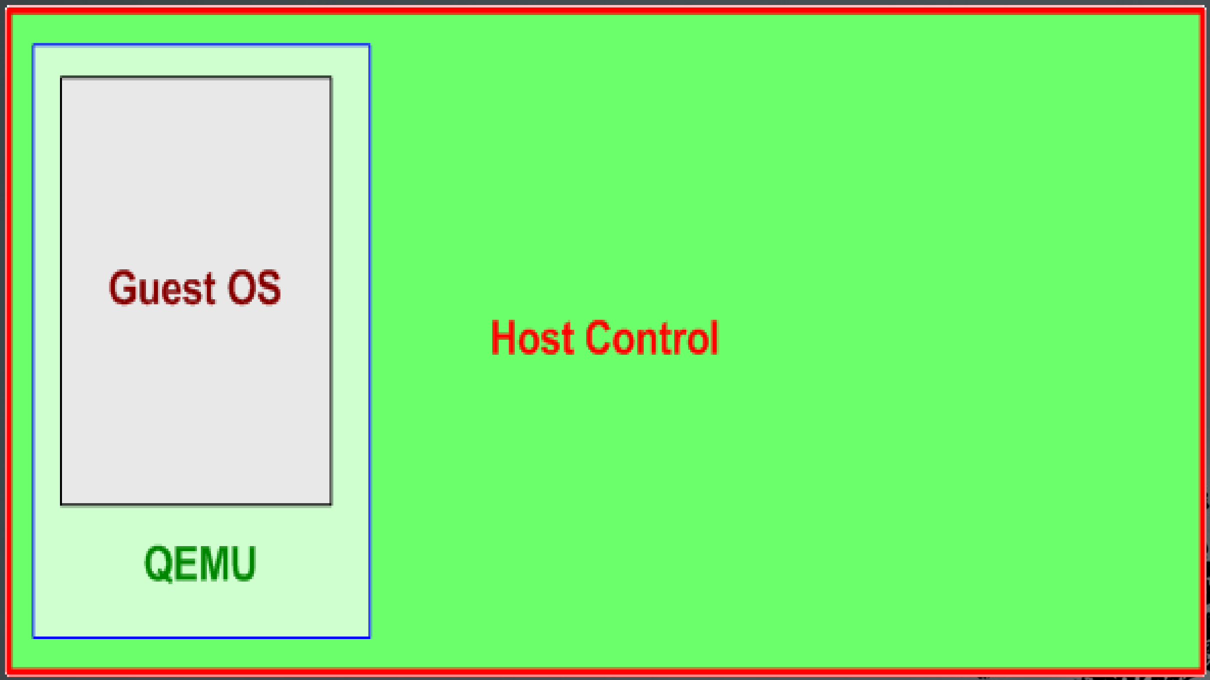
Xen Virtual Machine

- Host VM: Dom0
- Guest: DomU
  - Paravirtualized guest
  - Full-virtualized guest (HVM)



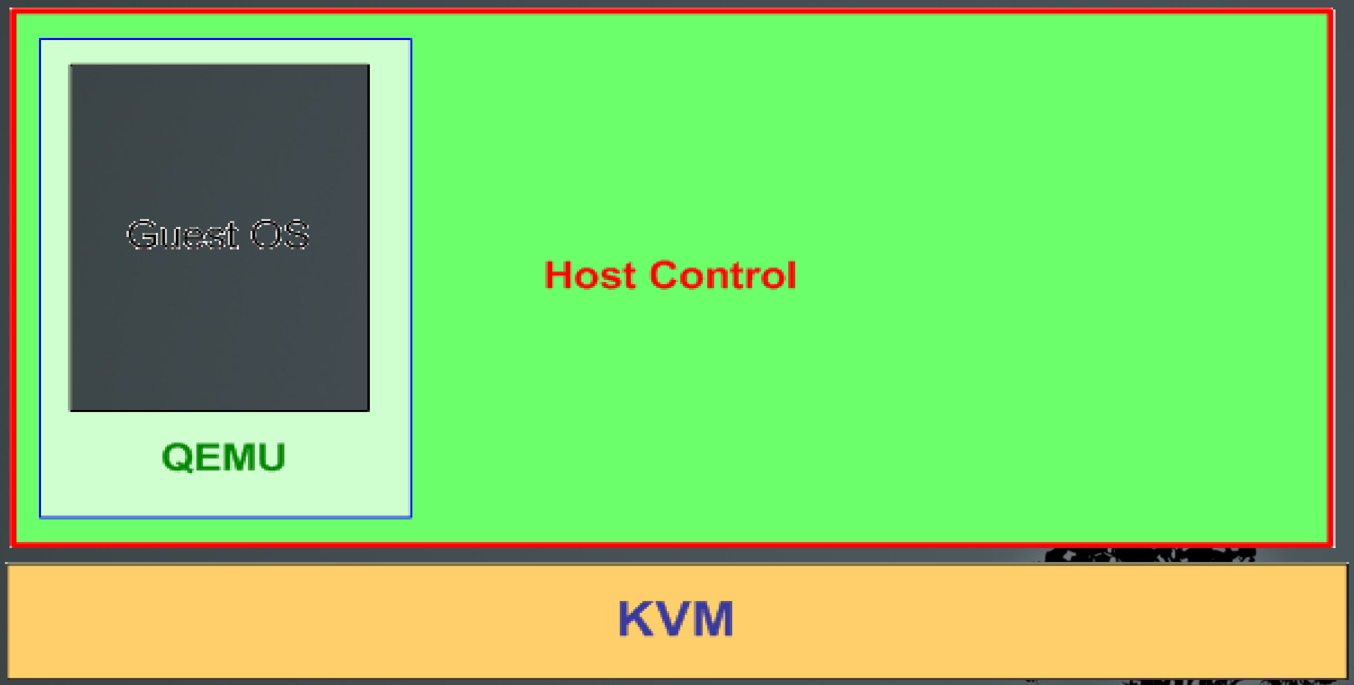
QEMU Emulator

- Host VM: Host OS
- Guest: QEMU process
  - Full-virtualized guest (HVM)



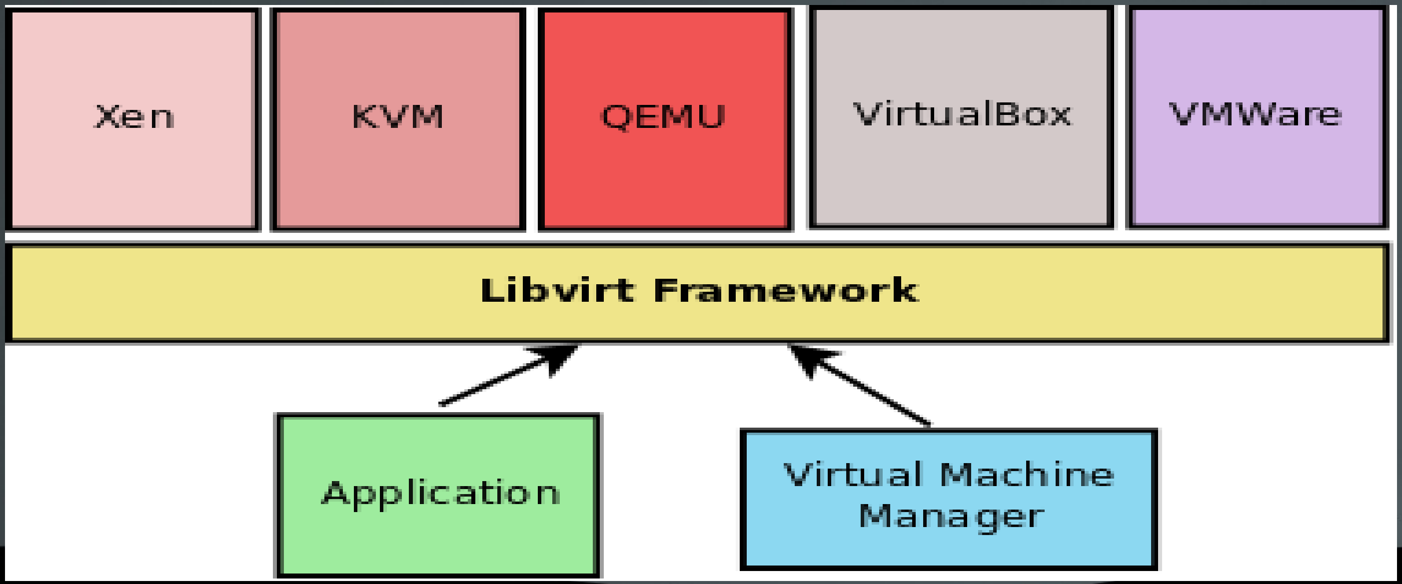
KVM Virtual Machine

- Host VM: Host OS
- Guest: KVM process
  - Full-virtualized guest (HVM)



Libvirt

- Provide a framework to access to VM!
  - VM-independent
  - Xen, KVM, QEMU, VirtualBox, VMWare supported
- Also include a toolkit to manage all kind of VM
- Become de-factor way to manage VM

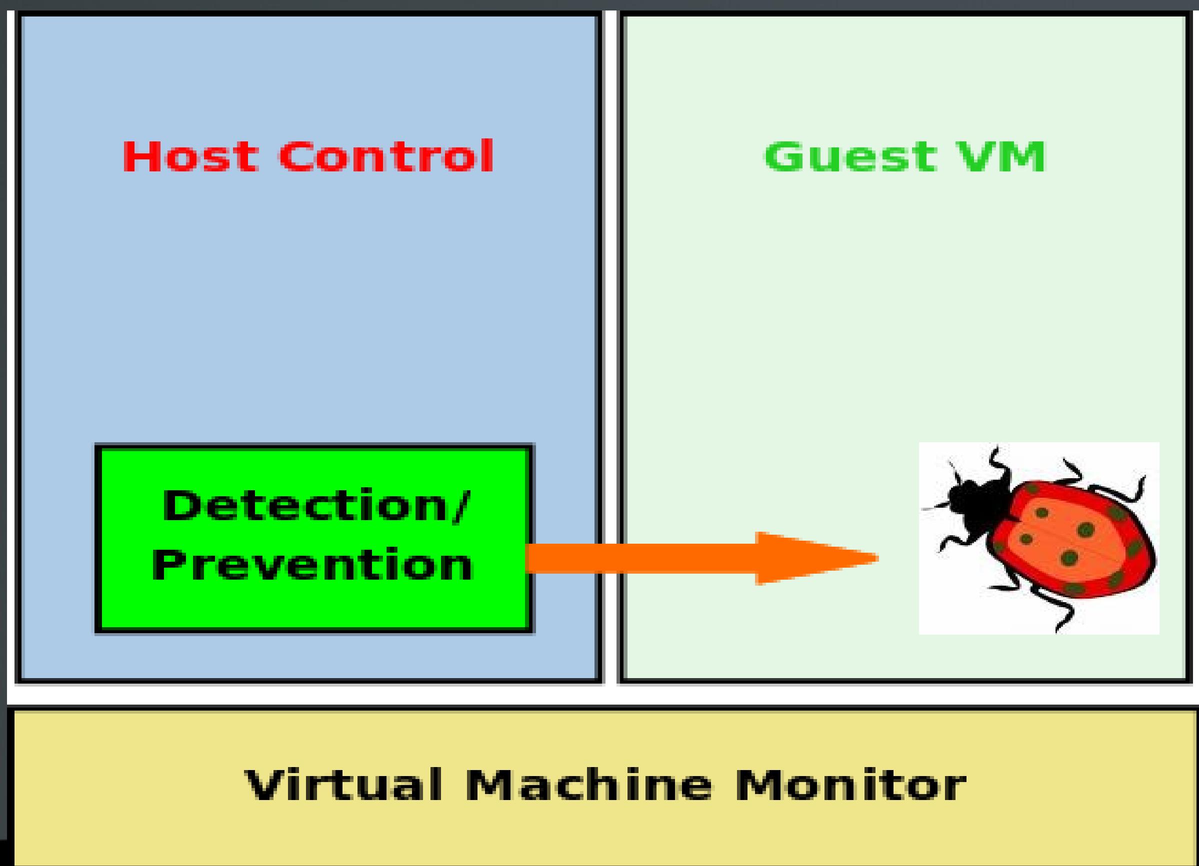


Detecting Malware for VM

- Put the scanner outside of protected VM
  - In privileged VM
- Let it access to VM's memory to perform different actions
  - Scan memory to detect malware
  - Manipulate memory (ie. write to) to disable malware

Rootkit Detector Architecture for VM

- Run the scanner in the privileged VM
- Access to protected VM thanks to VM interface



The Dream Comes True!

- This scanner satisfies all the dreamed requirements
  - Deal with memory-residence-only malware
  - Get the correct information, even if malware run at Operating System level
    - Does not rely on VM's OS to get information!
  - Very hard to be tampered, or disabled by malware
    - Impossible by design
- And even more!
  - Invisible to malware
  - Can effectively disable malware from outside

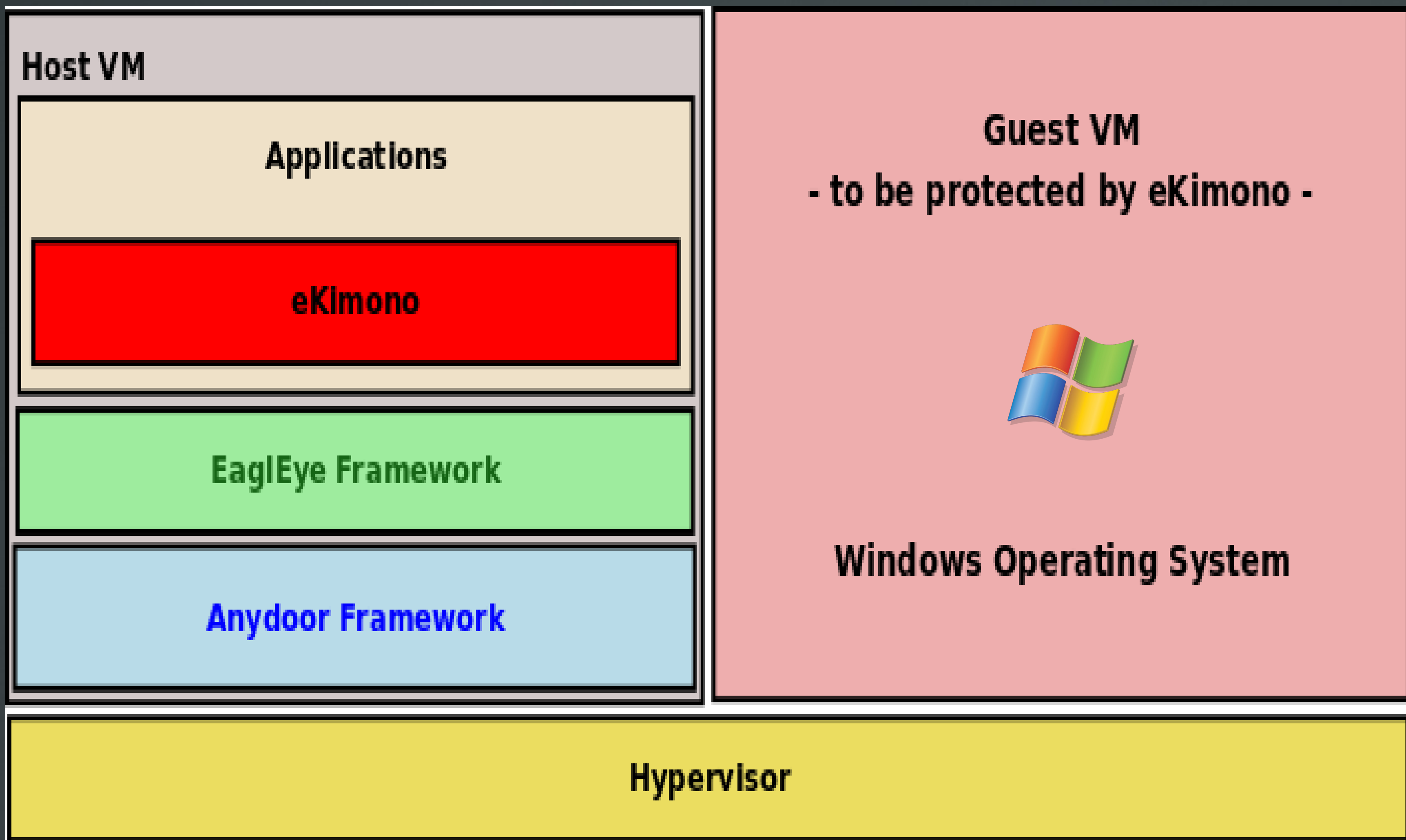
Challenges

- Analyzing raw memory to understand internal context of protected system
  - Understanding virtual memory
    - We have only physical memory access to VM
  - Retrieve OS-semantic objects correctly
    - Require excellent understandings on target's OS internals

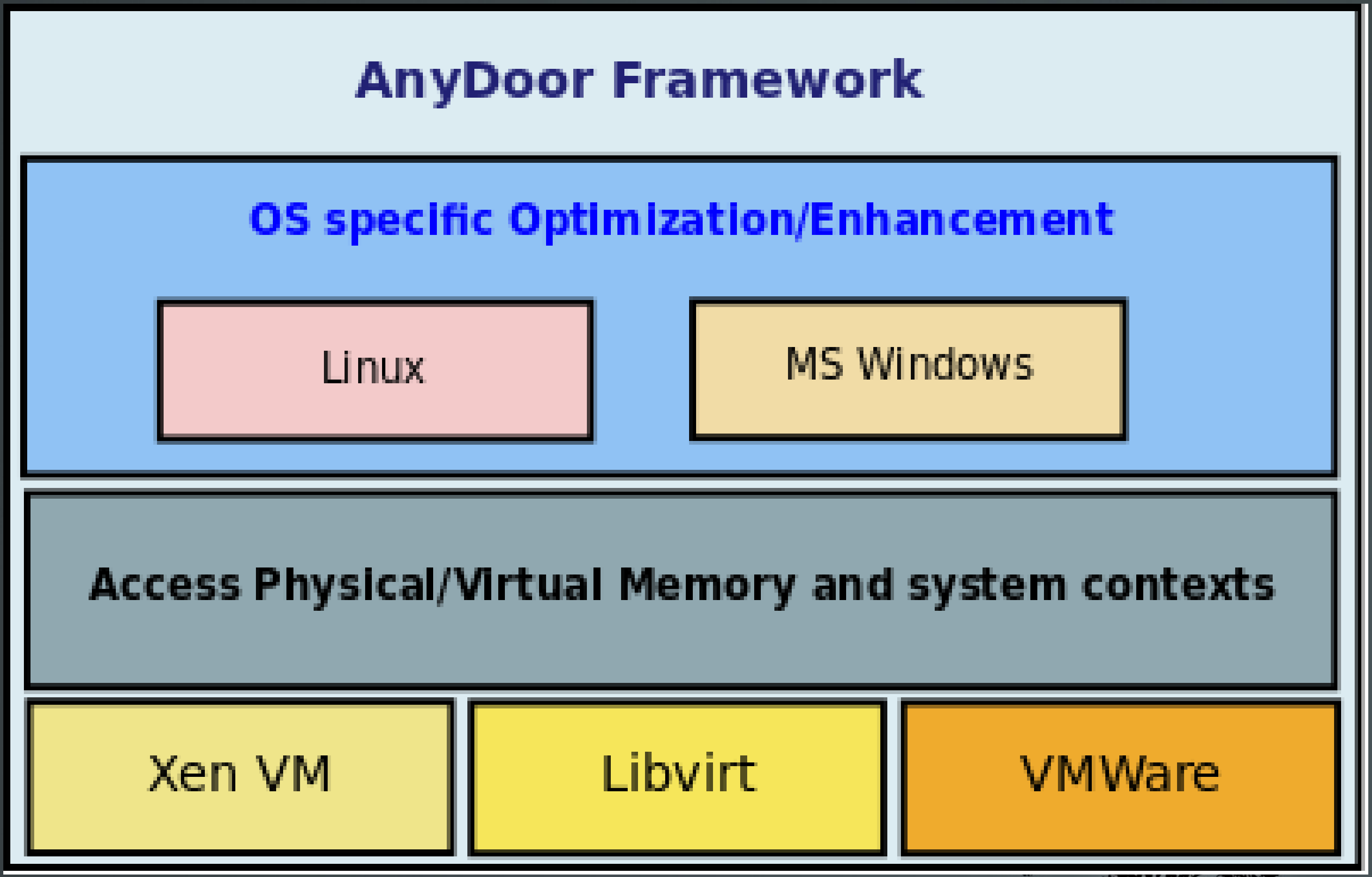
Multiple-layer Frameworks

- Understanding virtual memory
  - AnyDoor framework
- Retrieve OS-semantic objects
  - EaglEye framework

eKimono: Full Architecture



AnyDoor Architecture

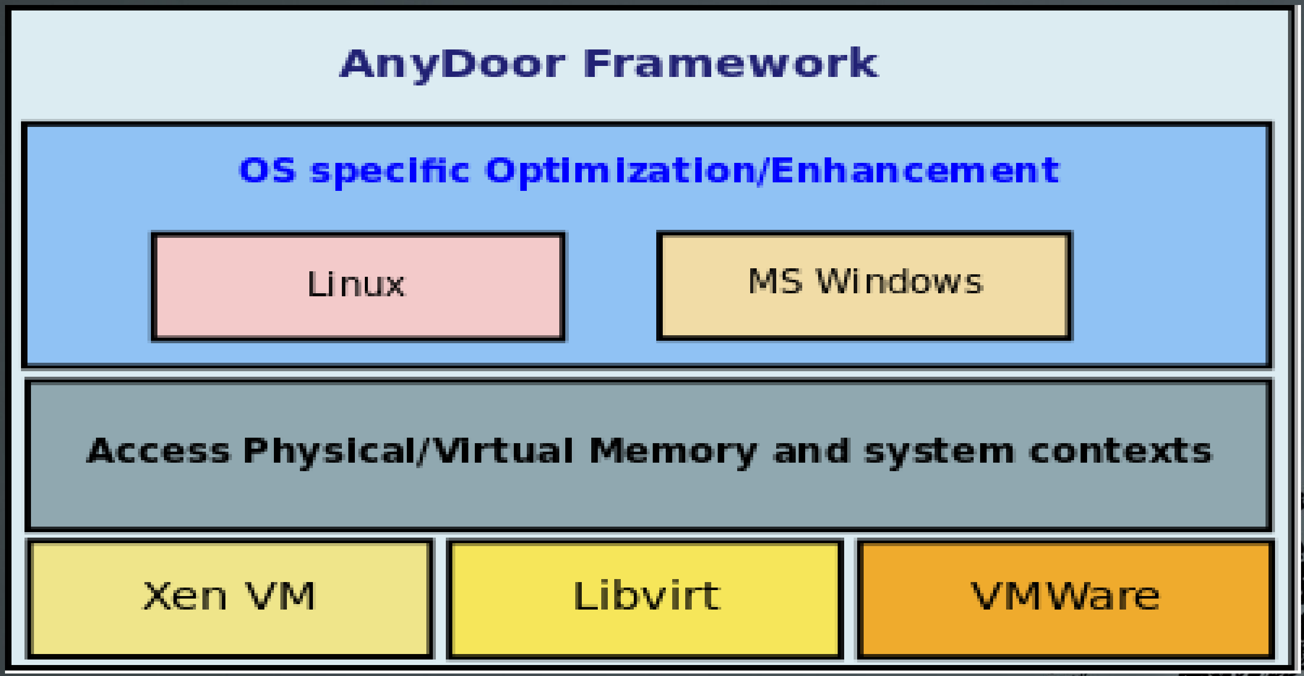


AnyDoor Framework

- Access to physical memory of protected system
  - OS independence
  - Target independence
    - **Xen**, **KVM**, **QEMU** supported so far
    - **VMWare** support is trivial, provided **VMSafe API** is public
- Provide access to virtual memory
  - Play a role of Memory-management-unit (MMU)
    - Software-based MMU
    - Must be able to understand all the memory mode (legacy/2MB pages/PAE,...)
- Provide access to registers of protected system

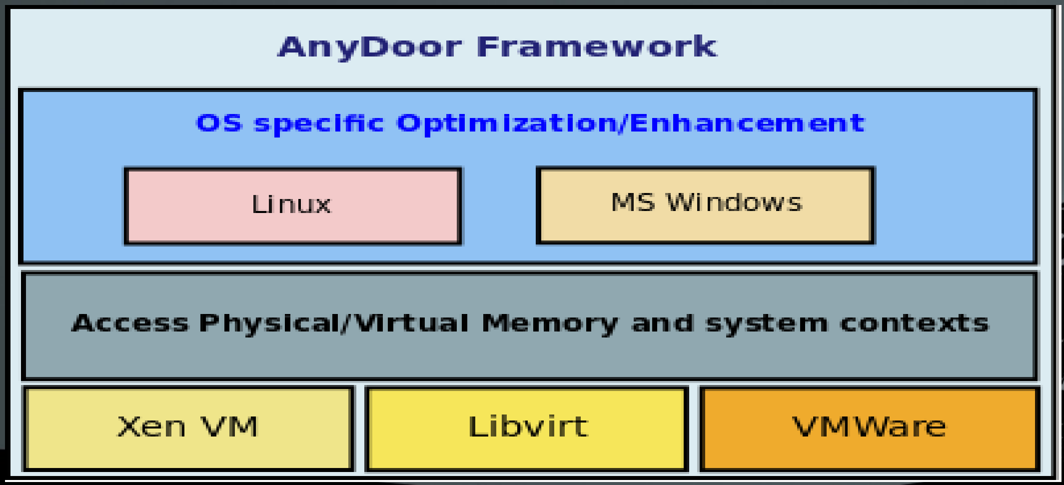
AnyDoor supports Xen

- Reimplement Xen's libxc functions to have access to DomU's physical memory & registers
  - LGPL license



AnyDoor Supports Libvirt

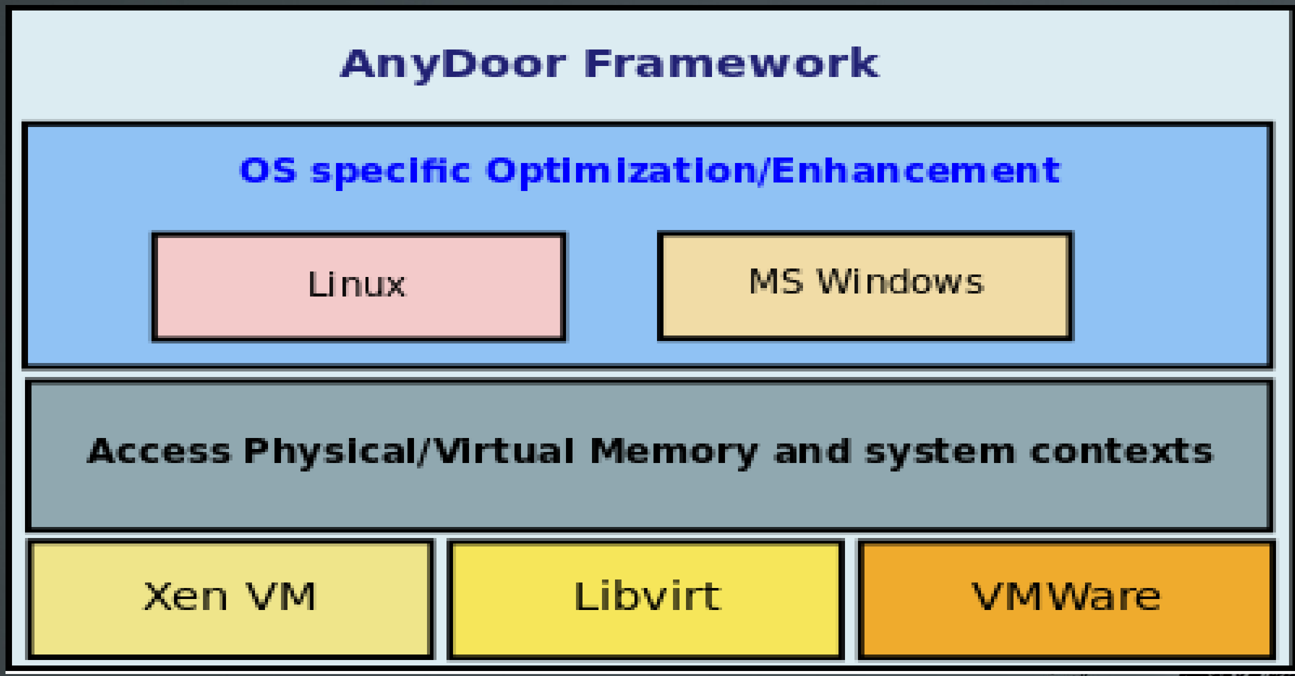
- Support KVM/QEMU via Libvirt interface
  - Take advantage of virtual memory access available in Libvirt
    - Speed up VM's virtual memory access
  - Patch Libvirt to support physical memory access
    - Patch accepted in Libvirt, and available from Libvirt 0.7 (5th August, 2009)



```
commit e4c48e02b4a7b4e49ee55ed62c65794d419a0b64
Author: Nguyen Anh Quynh <aquynh@gmail.com>
Date:   Wed Jul 22 16:27:09 2009 +0200

    Add support for physical memory access for QEmu

    * include/libvirt/libvirt.h include/libvirt/libvirt.h.in: adds the new
      flag VIR_MEMORY_PHYSICAL for virDomainMemoryPeek
    * src/libvirt.c: update the front-end checking
    * src/qemu_driver.c: extend the QEmu driver

diff --git a/include/libvirt/libvirt.h b/include/libvirt/libvirt.h
index 90007a1..86f56e5 100644
--- a/include/libvirt/libvirt.h
+++ b/include/libvirt/libvirt.h
@@ -619,6 +619,7 @@ int                    virDomainBlockPeek (virDomainPtr dom,
 /* Memory peeking flags. */
 typedef enum {
   VIR_MEMORY_VIRTUAL              = 1, /* addresses are virtual addresses */
+  VIR_MEMORY_PHYSICAL             = 2, /* addresses are physical addresses */
 } virDomainMemoryFlags;

 int                    virDomainMemoryPeek (virDomainPtr dom,
diff --git a/include/libvirt/libvirt.h.in b/include/libvirt/libvirt.h.in
index ba2b6f0..e6536c7 100644
--- a/include/libvirt/libvirt.h.in
+++ b/include/libvirt/libvirt.h.in
@@ -619,6 +619,7 @@ int                    virDomainBlockPeek (virDomainPtr dom,
 /* Memory peeking flags. */
 typedef enum {
   VIR_MEMORY_VIRTUAL              = 1, /* addresses are virtual addresses */
+  VIR_MEMORY_PHYSICAL             = 2, /* addresses are physical addresses */
 } virDomainMemoryFlags;

 int                    virDomainMemoryPeek (virDomainPtr dom,
diff --git a/src/libvirt.c b/src/libvirt.c
index 7463c06..0e6d88f 100644
--- a/src/libvirt.c
+++ b/src/libvirt.c
```

AnyDoor Supports VMWare

- Trivial to support VMWare if we have access to VM's physical memory
  - Exactly what is provided by VMSafe API
    - Still close to public now!



Sample AnyDoor API

```
/* <anydoor/anydoor.h> */

/* Read data from memory of a process running inside a target. */
int anydoor_read_user(anydoor_t h, unsigned long pgd, unsigned long vaddr,
        void *buf, unsigned int size);

/* Write data into memory of a process running inside a target. */
int anydoor_write_user(anydoor_t h, unsigned long pgd, unsigned long addr,
        void *buf, unsigned int size);

/* Read data from a target's physical memory. */
int anydoor_read_pa(anydoor_t h, unsigned long paddr, void *buf, unsigned int
    size);

/* Write data into a target's physical memory. */
int anydoor_write_pa(anydoor_t h, unsigned long paddr, void *buf, unsigned int
    size);
```

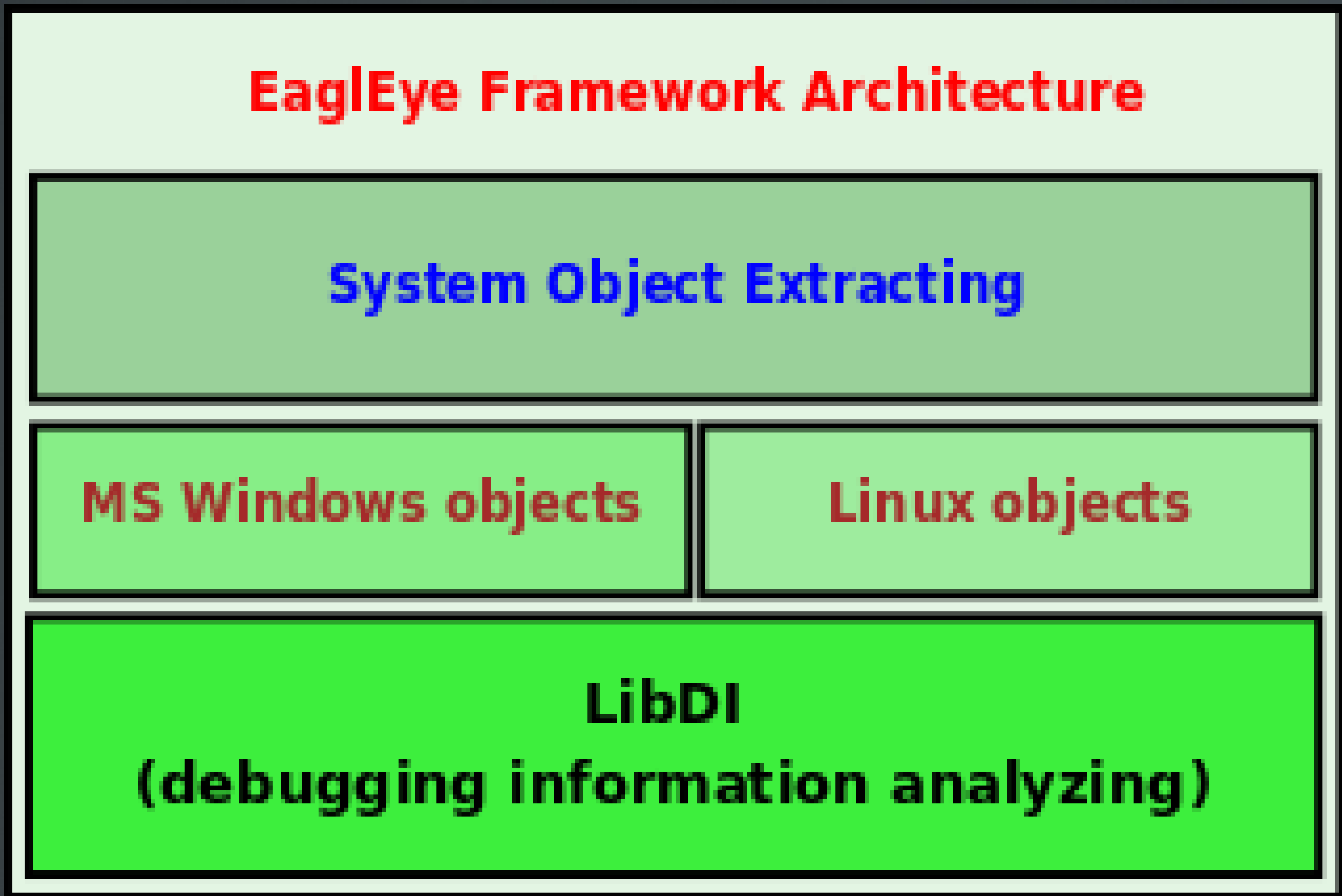
Challenges

- Analyzing raw memory to understand internal context of protected system
  - Understanding virtual memory
  - Retrieve OS-semantic objects
    - Require excellent understandings on target's OS internals

EaglEye Framework

- Use the service provided by AnyDoor to access to virtual/physical memory of protected system
- Retrieve OS-objects from virtual/physical memory of guest VM
  - Focus on important objects, especially which usually exploited by malware, or can disclose their residence
    - Network ports, connections
    - Processes
    - Kernel modules
    - etc...

EaglEye Architecture



Challenges

- Retrieve semantic objects requires excellent understanding on OS internals
  - Locate the objects
  - Actually retrieve objects and its internals
    - How the objects are structured?
      - Structure size?
      - Structure members?
      - Member offset?
      - Member size?
      - ...

Locate OS's Objects

- Kernel modules
- Processes/threads
- System handles
- Open files
- Registries
- DLLs
- Network connections/ports
- Drivers, symbolic links, ...

Retrieve Objects

- Must understand object structure
  - Might change between Windows versions, or even Service Pack

```
struct _EPROCESS {
    KPROCESS Pcb;                              → offset 0, size 0x6c
    EX_PUSH_LOCK ProcessLock;→ offset 0x6c, size 4
    LARGE_INTEGER CreateTime;         → offset 0x70, size 8
    LARGE_INTEGER ExitTime;              → offset 0x78, size 8
    EX_RUNDOWN_REF RundownProtect;      → offset 0x80, size 4
    ....
```

Current Solutions?

- Hardcode all the popular objects, with offsets & size of popular fields?
  - Does by everybody else
  - But this is far from good enough!
    - Limited to objects you specify
    - Limited to only the offsets you specify

Another Dream ...

- To be able to query structure of all the objects, with their fields
  - Support all kind of OS, with different versions
  - On demand, at run-time, with all kind of objects
  - Various questions are possible
    - What is the size of this object?
    - What is the offset of this member field in this object?
    - What is the size of this array member of this object?
    - ...

... Comes True, Again: LibDI

- Satisfy all the above requests, and make your deam come true
  - Come in a shape of another framework
  - Rely on public information on OS objects
    - OS independence
      - Windows and Linux are well supported so far
    - Have information in debugging formats DWARF , and extract their structure out at run-time

Windows Objects

- ReactOS file header prototypes
  - Free & open to public
  - Support Win2k3 and up.
  - Compile ReactOS file header prototypes with debugging information

g++ -g windows.c -c -o <windows_VERSION.o>

Windows Objects - Problems

- ReactOS only supports Win2k3 and up
- Need to patch headers to support WinXP and prior versions
  - From Windows debugging symbols data
  - Patch size is small
- Fix incorrect and not updated data structures
  - Windows Vista, Windows 2008
- Patch to support recent Windows OS, like Windows 7

Sample LibDI API

```
/* <libdi/di.h> */

/* Get the struct size, given its struct name */

int di_struct_size(di_t h, char *struct_name);

/* Get the size of a field of a struct, given names of struct and member. */

int di_member_size(di_t h, char *struct_name, char *struct_member);

/* Get the offset of a field member of a struct, given names of struct and member */

int di_member_offset(di_t h, char *struct_name, char *struct_member);
```

Sample Code using LibDI

```
#include <libdi/di.h>
...
di_t h;
di_open("windows_WINXPSP3.o", &h);
// size of _EPROCESS
int s1 = di_struct_size(h, "_EPROCESS");
// size of _EPROCESS::CreateTime
int m1 = di_member_size(h, "_EPROCESS", "CreateTime");
// offset of _EPROCESS::CreateTime
int o1 = di_member_offset(h, "_EPROCESS", "CreateTime");
di_close(h);
```

EaglEye: Retrieve Objects

- Separate API for each kind of objects
- Designed so it is hard to be abused or tampered by guest VM
  - Get first object in the list of objects
    - Usually the head of object list must be located
    - Or by scanning the pool memory, or scanning in physical memory
      - Using pattern-matching technique
  - Get next objects
  - One by one, until reach the last object

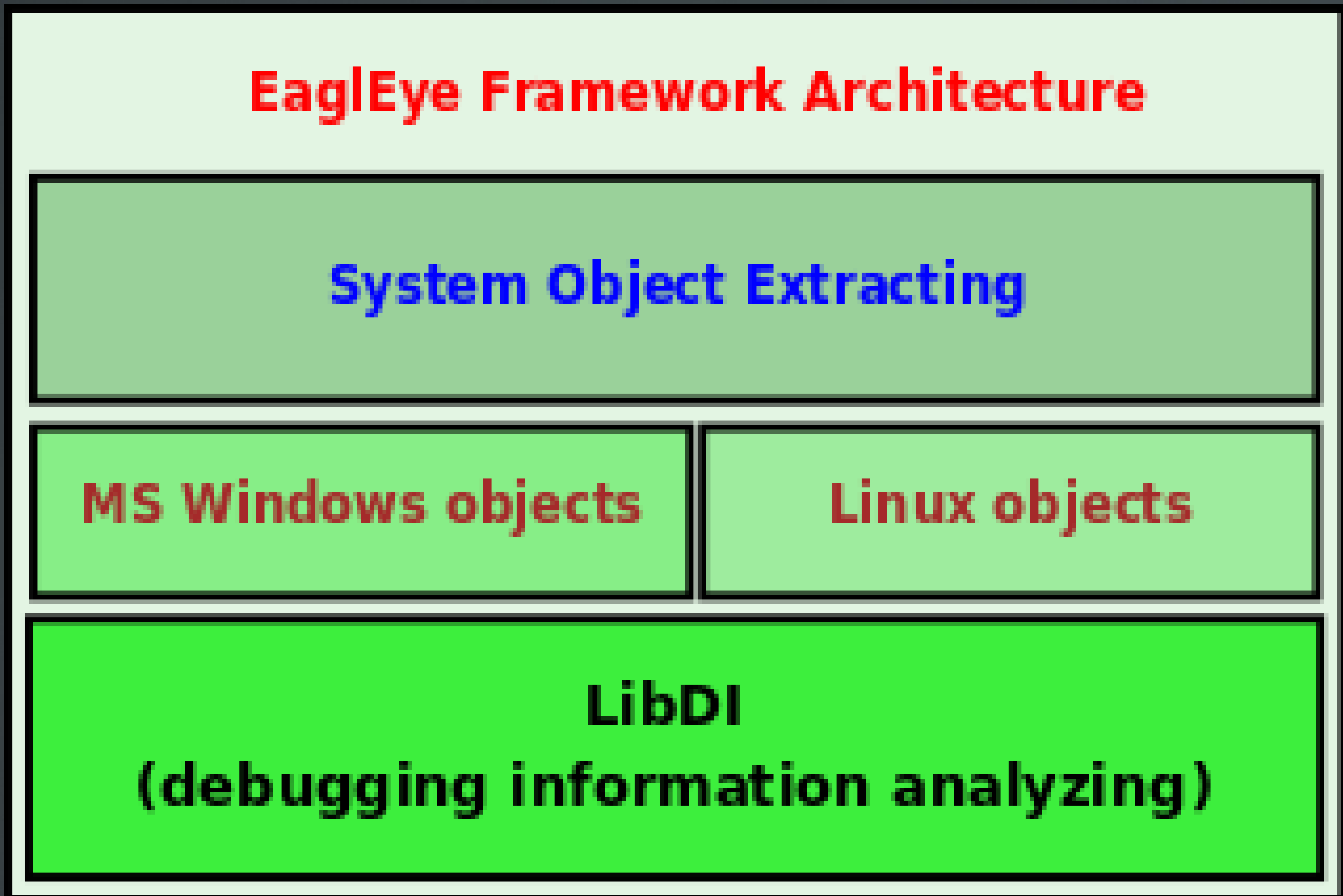
Sample EaglEye API (1)

```
/* <eagleye/eagleye.h> */

/* @task: output value, pointed the the kernel memory keep task info */
int ee_get_task_first(ee_t h, unsigned long *task);

/* @task: output value, pointed the the kernel memory keep task info */
int ee_get_task_next(ee_t h, unsigned long *task);

/* get the pointer to the process struct, given the process's pid.
int ee_get_task_pid(ee_t h, unsigned long pid, unsigned long *task);

/* get the first open dll file of a task with a given process id.
 * on return, dll points to the userspace memory that keeps dll info */
int ee_get_task_dll_first(ee_t h, unsigned long pid, unsigned long *dll);

/* get the next open dll file of a task with a given process id.
int ee_get_task_dll_next(ee_t h, unsigned long *dll);
```

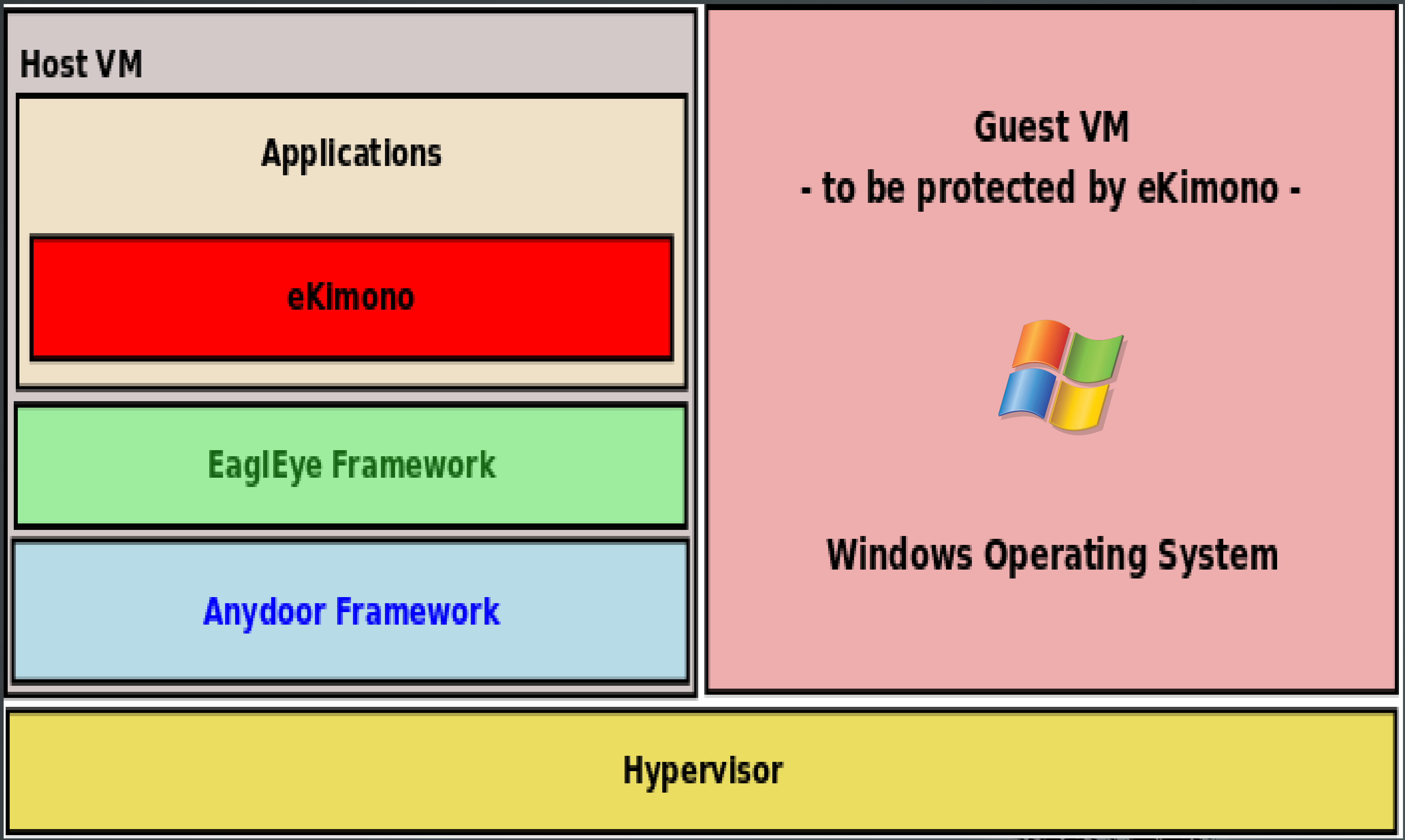
Sample EaglEye API (2)

```
/* <eagleye/windows.h> */

/* get process image filename, given its EPROCESS address */
int windows_task_imagename(ee_t h, unsigned long eprocess, char *name,
    unsigned int count);

/* get process id, given its EPROCESS address */
int windows_task_pid(ee_t h, unsigned long eprocess, unsigned long *pid);

/* get parent process id, given its EPROCESS address */
int windows_task_ppid(ee_t h, unsigned long eprocess, unsigned long
    *ppid);

/* get process cmdline, given its EPROCESS address */
int windows_task_cmdline(ee_t h, unsigned long eprocess, char *cmdline,
    unsigned int count);
```

EaglEye Architecture

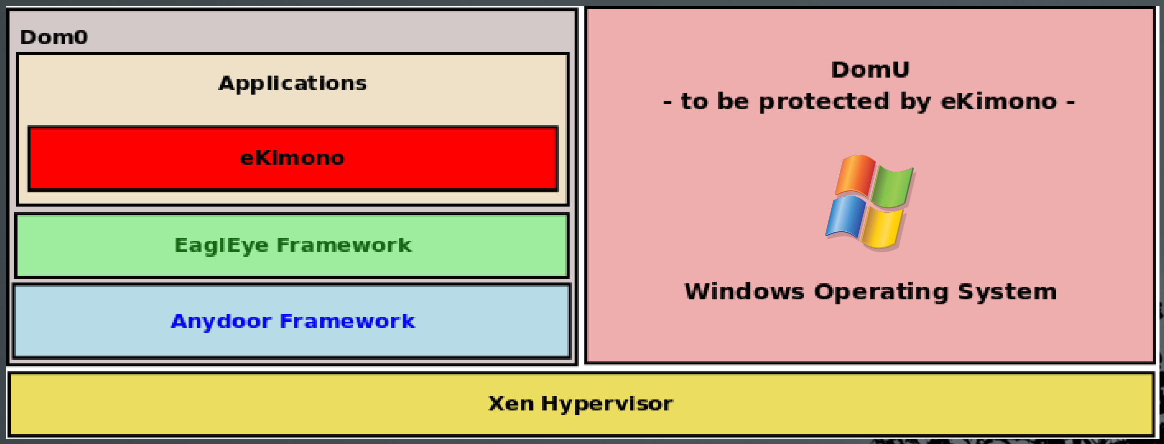


eKimono Architecture



eKimono: Other Advantages

- Require absolutely no support from protected VM
  - No agent is required!
- Zero-cost deployment
  - Just run the scanner side protected VM



eKimono Implementation

- Use the service provided by EaglEye to OS-objects
- Perform various tactics to detect malware
  - Baseline-based detection
  - Cross-view detection
  - Black/White list checking
  - Abnormal behaviour detection

Baseline-based Detection

- Detect the malware that modify the baseline information
    - Define the baseline of clean system
        - Kernel modules
            - List of modules with attributes
            - Hashing values of text area
            - Etc...
        - System calls
        - Processes
            - DLLs, imported functions
        - Network connections/ports

Cross-view Detection

- Compare critical system objects from different point of views to find hidden objects (rootkits?)
  - Process
  - List of processes
  - Threads
  - DLLs
  - Kernel modules
  - Network connections/ports

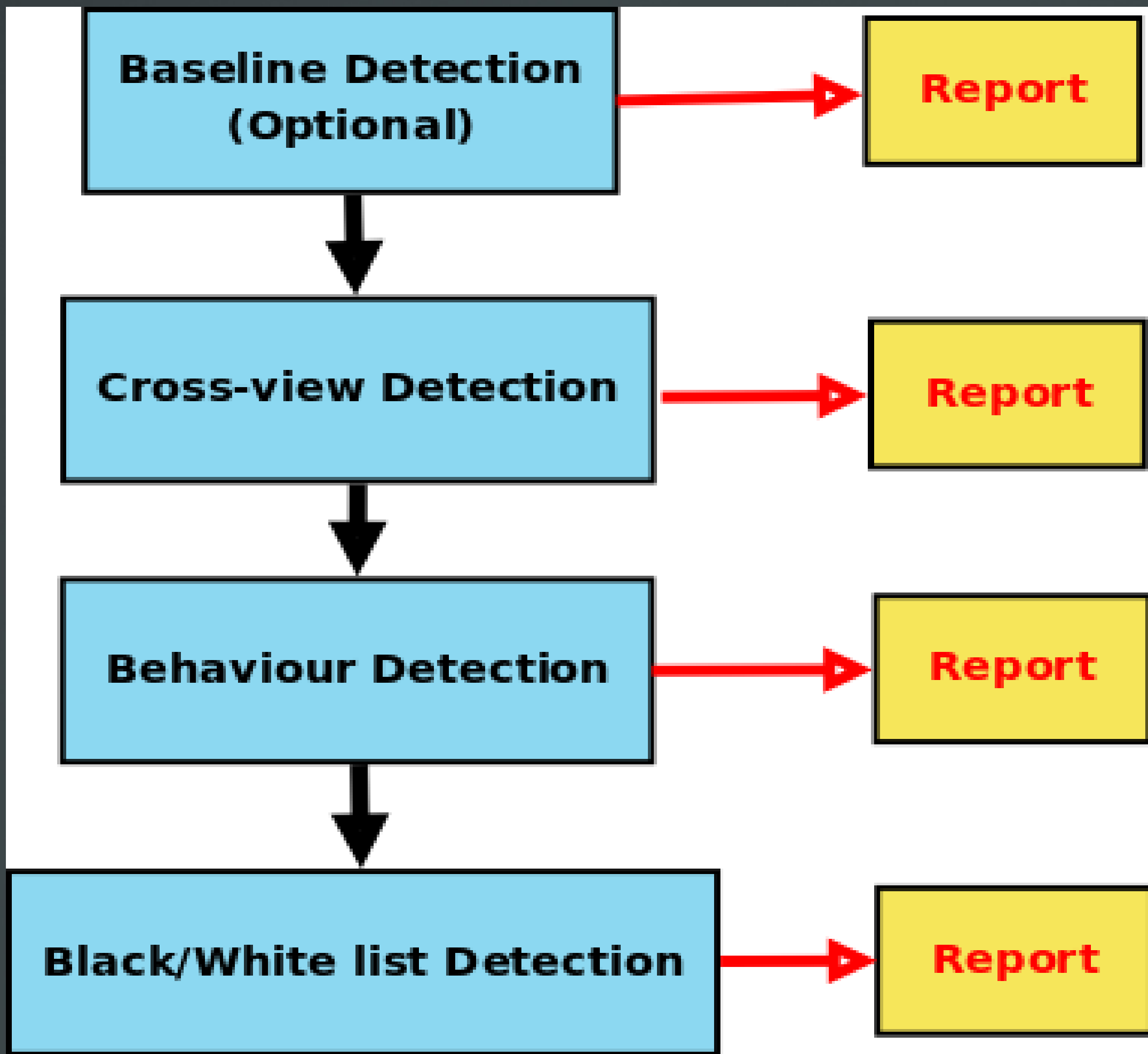
Black/White List Checking

- Black-list detection
  - List of known-bad objects
    - Based on names, hashing values, …
    - Process
    - Kernel modules
    - Network ports
- White-list checking
  - Exceptions that should not be reported
  - Eliminate false-possitives

Abnormal Behavior Detection

- Abnormal behaviours
  - Modification to critical places?
    - Process IAT/EAT
    - EAT of critical modules & DLLs
    - System calls
    - Kernel driver IRQ
    - IDT, GDT
    - ...

Detection Model



Part III

- Problems of current malware scanner
- eKimono: Malware scanner for Virtual Machines
  - Introduction on Virtual machine
  - Architecture, design and implementation of eKimono
  - Focus on Windows protection
- **eKimono demo on detecting malware**
- Conclusions

Detecting Rootkits

- Userspace rootkits
- Kernel rootkits

Project Status

- Under heavy development
- Around 30.000 lines of code totally
  - Frameworks & scanner
  - Good support for Windows XP
- In-progress work
  - To support other editions of Windows
    - Windows Vista, Windows 7
  - Linux support

Part IV

- Problems of current malware scanner
- eKimono: Malware scanner for Virtual Machines
  - Introduction on Virtual machine
  - Architecture, design and implementation of eKimono
  - Focus on Windows protection
- eKimono demo on detecting malware
- **Conclusions**

Conclusions

- Put the malware scanner outside of protected system has many advantages
  - Zero-cost on deployment
  - Tamper resistant to malware inside the VM
  - Invisible to malware
  - Can mitigate damages effectively from outside

eKimono: Detecting rootkits inside Virtual Machines

Q & A

Nguyen Anh Quynh

<aquynh@gmail.com>