



---

# Cloud Computing Vulnerability

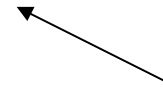
Fabrizio Baiardi  
f.baiardi@unipi.it



# Syllabus

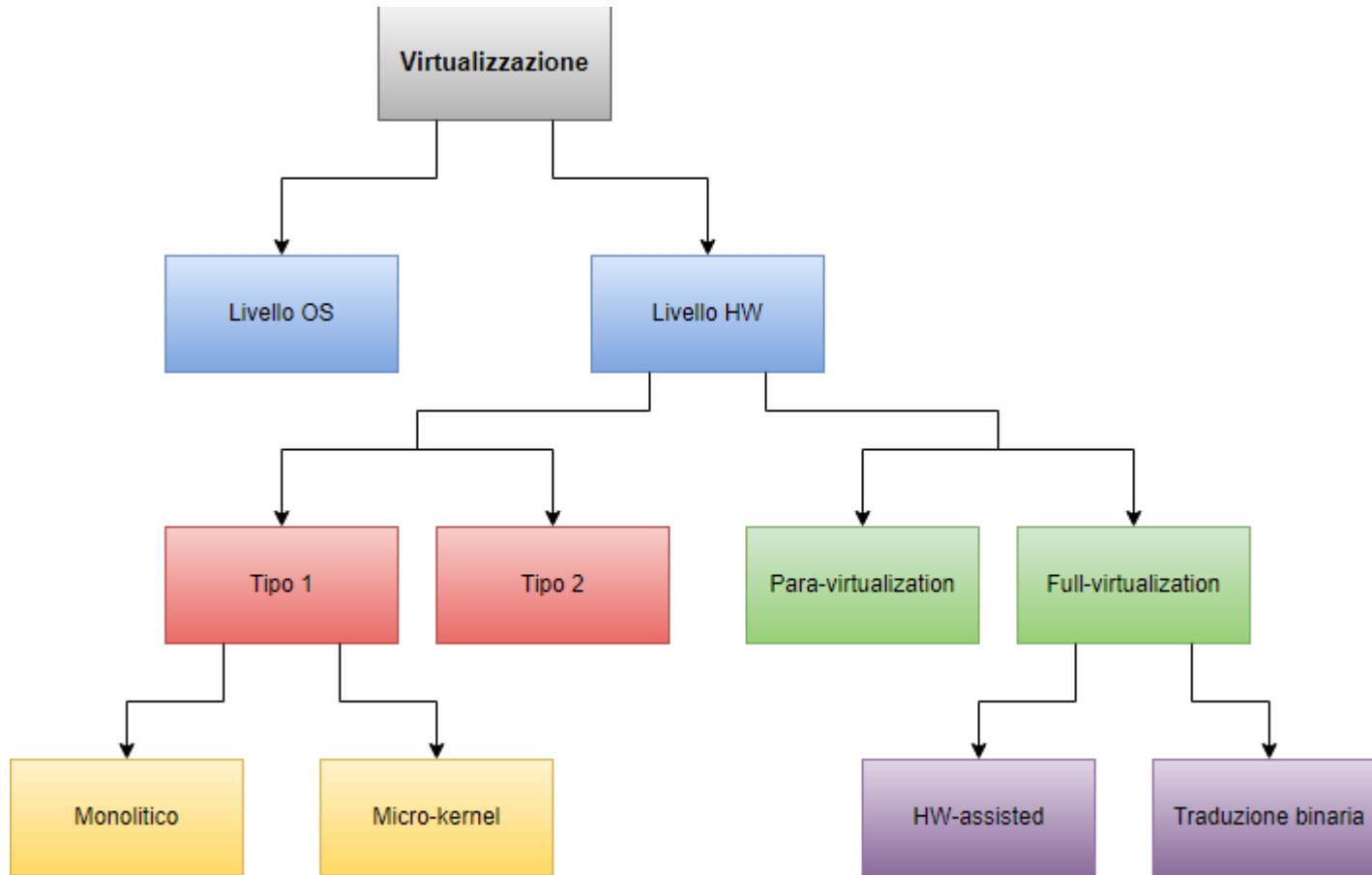
---

- ICT Security
- Cloud Computing
- Security of Cloud Computing
  - New Threat Model
  - New Attacks
  - Countermeasures



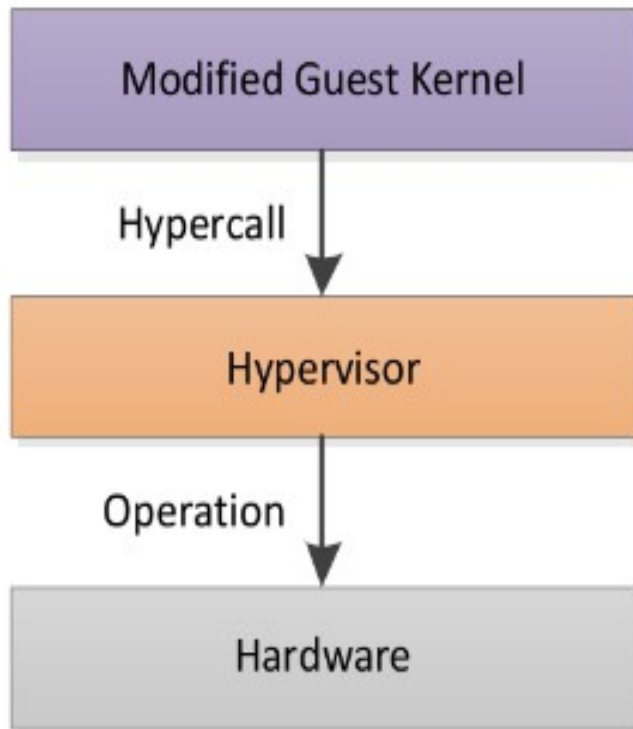
**A first set of VM and VMMM  
Vulnerabilities and of attacks**

# Virtualization Solutions

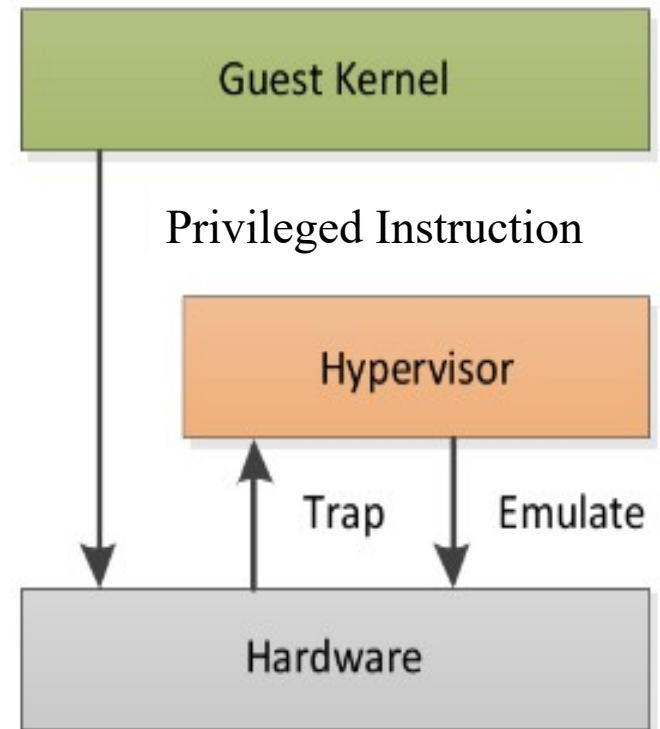


# Emulation

## Para-virtualization



## "Classical" Full-virtualization





# Origins - Principles

---

Instruction types (security some time ago :-D )

Privileged vs unprivileged instructions and modes

an instruction traps in unprivileged (user) mode but not in privileged (supervisor) mode.

Sensitive

- Control sensitive – attempts to change the memory allocation or privilege mode
- Behavior sensitive
  - Location sensitive – execution behavior depends on memory location
  - Mode sensitive – execution behavior depends on privilege mode

Innocuous – an instruction that is not sensitive

Theorem

For any conventional computer, a virtual machine monitor may be built if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Significance = The IA-32/x86 architecture is not virtualizable.

---



# Classical solution

---

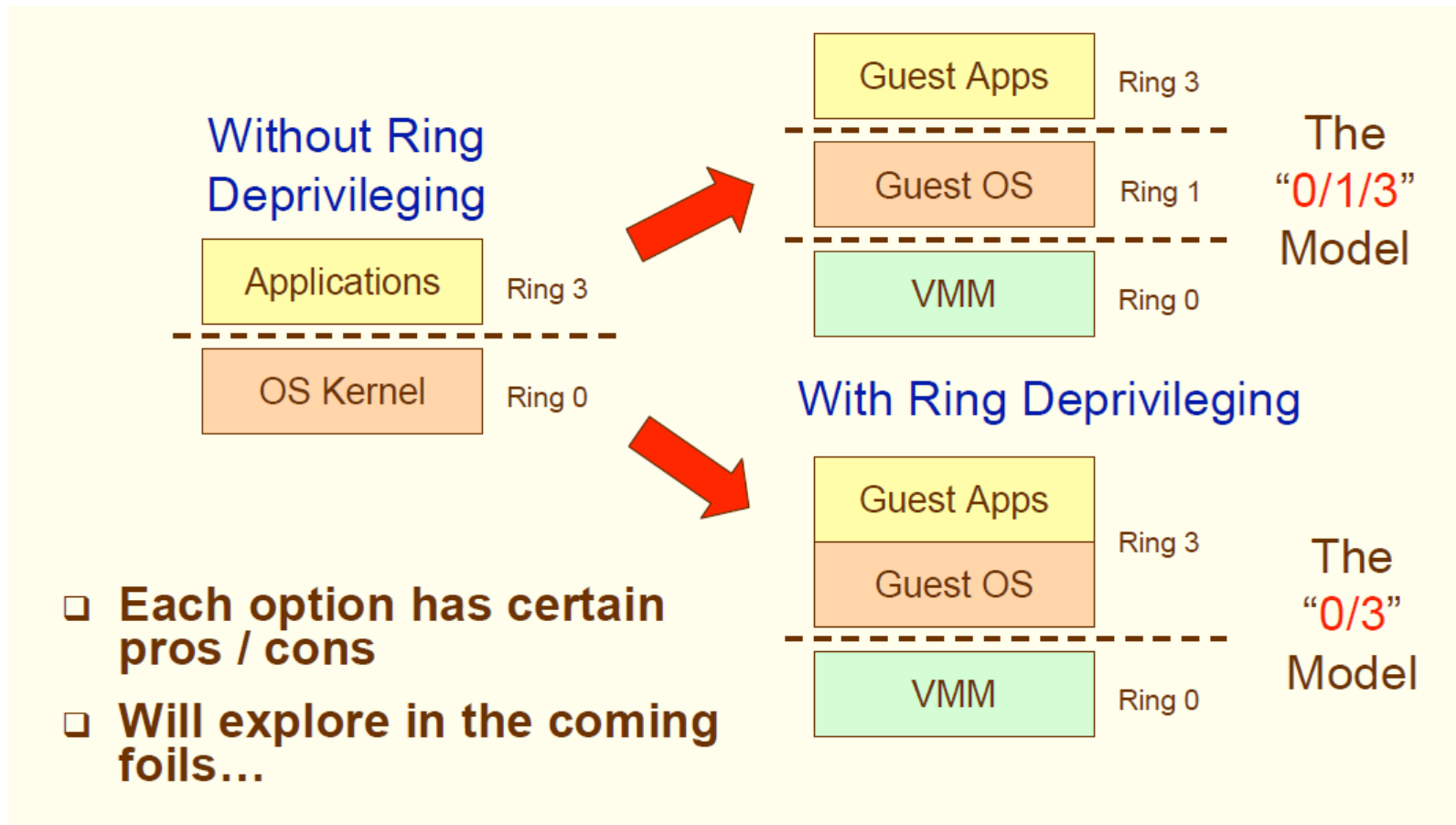
## Trap and Emulate

- Run guest operating system *deprivileged*
- All privileged instructions *trap into VMM*
- VMM emulates instructions against virtual state  
*e.g. disable virtual interrupts, not physical interrupts*
- Resume direct execution from next guest instruction

## Implementation Technique

- This is just one technique
- Popek and Goldberg criteria permit others

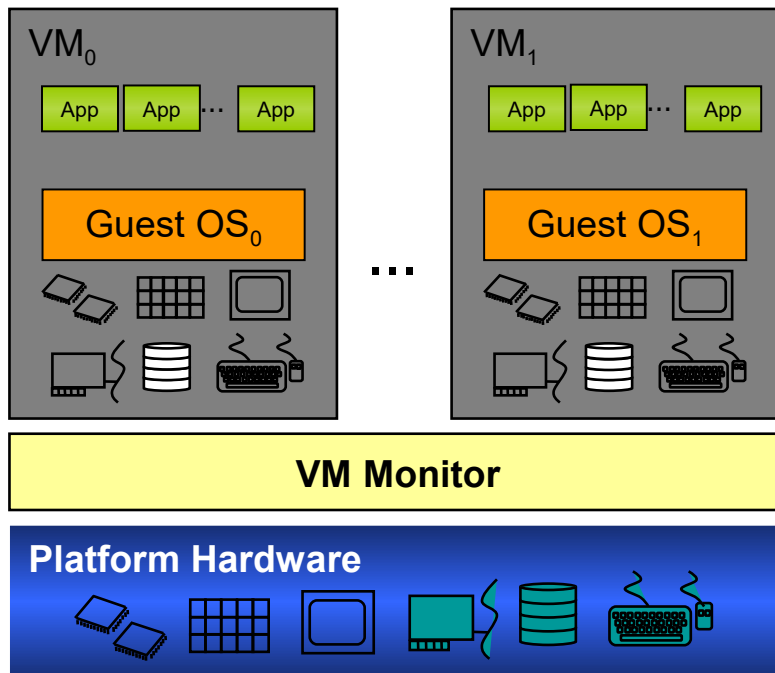
# Some Options



# Solution for IA-32 arch

## Ring Deprivileging =

- all guest software should be run at a privilege level greater than 0.
- privileged instructions generate faults = VMM runs in Ring-0 as a collection of fault handlers.
- the guest OS should not be able to update the VMM



- The VMM interprets in software privileged instructions that would be executed by an OS.
- Any non privileged instruction issued by an OS or Application Environment is executed directly by the machine.
- A guest OS could be deprivileged in two distinct ways:
  - it could run either at privilege level 1 (the 0/1/3 model) or ,
  - It could run at privilege level 3 (the 0/3/3 model).





# Virtualization challenges

## Ring Aliasing

Problems if software is run at a privilege level other than the privilege level for which it was written.

- The CS register points to the code segment.
- If the *PUSH* is executed with the CS register, the register content (with the current privilege level) is pushed on the stack



A guest OS could easily determine that it is not running at privilege level 0.

## Address-Space Compression

OSs expect to have access to the processor's full virtual address space (in IA-32. linear address space)

- The VMM could run entirely within the guest's virtual-address space but it would use a substantial amount of the guest's virtual address space.
- The VMM could run in a separate address space, but it must use a minimal amount of the guest's virtual address space to manage transitions between guest software and the VMM (IDT and GDT for IA-32)

To preserve its integrity, the VMM must prevent guest access to those portions of the guest's virtual address space that it is using.



# Virtualization challenges

## Excessive Faulting

Ring deprivileging interferes with the effectiveness of facilities in the IA-32 architecture that accelerate the delivery and handling of transitions to OS software.

- The IA-32 *SYSENTER* and *SYSEXIT* instructions support low-latency system calls.
- *SYSENTER* always effects a transition to privilege level 0, and *SYSEXIT* faults if executed outside that ring



With VMM it does traps to the OS but to the VMM that emulates every execution of *SYSENTER* and *SYSEXIT* to implement interactions with the OS causing serious performance problems.

## Non-Trapping Instructions

Some instructions access privileged state and do not fault when executed with insufficient privilege.

- the IA-32 registers GDTR, IDTR, LDTR, and TR contain pointers to data structures that control CPU operation. Software can execute the instructions that read, or store, from these registers at any privilege level.



# Virtualization challenges

---

## Interrupt Virtualization

- The mechanisms of masking external interrupts for preventing their delivery when the OS is not ready for them is a big challenge for the VMM design.
- The VMM must manage the interrupt masking in order to prevent an OS from masking the external interrupts because this prevents any guest OS to receive interrupts.
  - IA-32 uses the interrupt flag (IF) in EFLAGS register to control interrupt masking. IF= 0 interrupts are masked.

## Access to Hidden State

- Some components of the processor state are not represented in any software-accessible register.
  - the IA-32 has the hidden descriptor caches for segment registers. A segment-register load copies of the GDT and LDT into this cache, which is not modified if software later writes to the descriptor tables.



# Virtualization challenges

---

## Ring Compression

Ring depriving uses privilege-based mechanisms to protect the VMM from guest software. IA-32 includes two mechanisms: segment limits and paging:

- Segment limits do not apply in 64-bit mode.
- Paging must be used.
  - Problem: IA-32 paging does not distinguish privilege levels 0-2.
    - » The guest OS must run at privilege level 3 (the 0/3/3 model).
    - » The guest OS is not protected from the guest applications.

## Frequent Access to Privileged Resources

The performance is compromised if the privileged resources are accessed too many times generating too many faults that must be intercepted by the VMM.

- For example: the task-priority register (TPR), in IA-32 located in the advanced programmable interrupt controller (APIC), is accessed with very high frequency by some OSs.



# Alternative solutions

---

## Interpretation

- Problem – too inefficient
- x86 decoding slow

## Code Patching

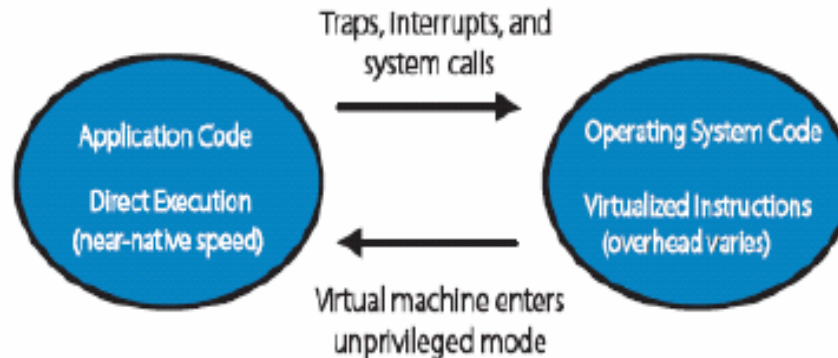
- Problem – not transparent
- Guest can inspect its own code

## Binary Translation (BT)

- Approach pioneered by VMware
- Run any unmodified x86 OS in VM

## Extend x86 Architecture

# Software VMM



Direct execute unprivileged guest application code

- Will run at full speed until it traps, we get an interrupt, etc.

“Binary translate” all guest kernel code, run it unprivileged

- Since x86 has non-virtualizable instructions, *proactively* transfer control to the VMM (no need for traps)
- Safe instructions are emitted without change
- For “unsafe” instructions, emit a controlled emulation sequence
- VMM translation cache for good performance



# Solution adopted by VMware

---

**Binary** – input is x86 “hex”, not source

**Dynamic** – interleave translation and execution

**On Demand** – translate only what about to execute (lazy)

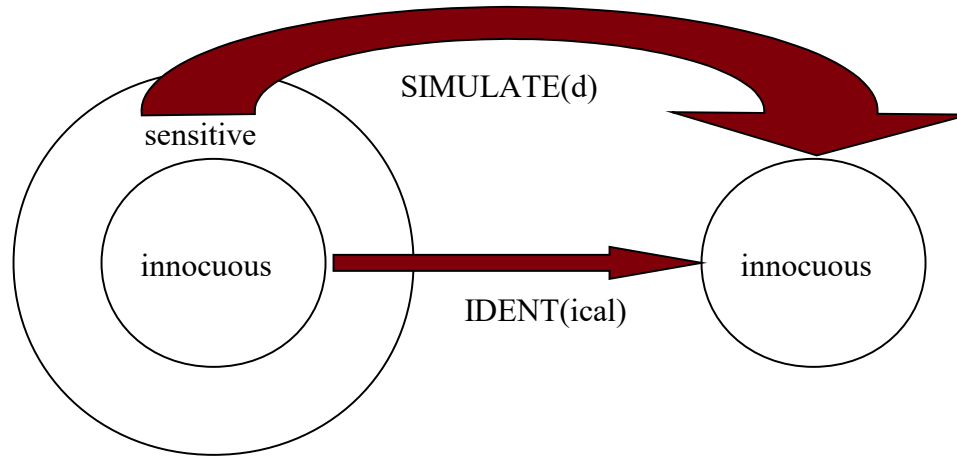
**System Level** – makes no assumptions about guest code

**Subsetting** – full x86 to safe subset

**Adaptive** – adjust translations based on guest behavior

# Binary Translation

---



## Characteristics

Binary – input is machine-level code

Dynamic – occurs at runtime

On demand – code translated when needed for execution

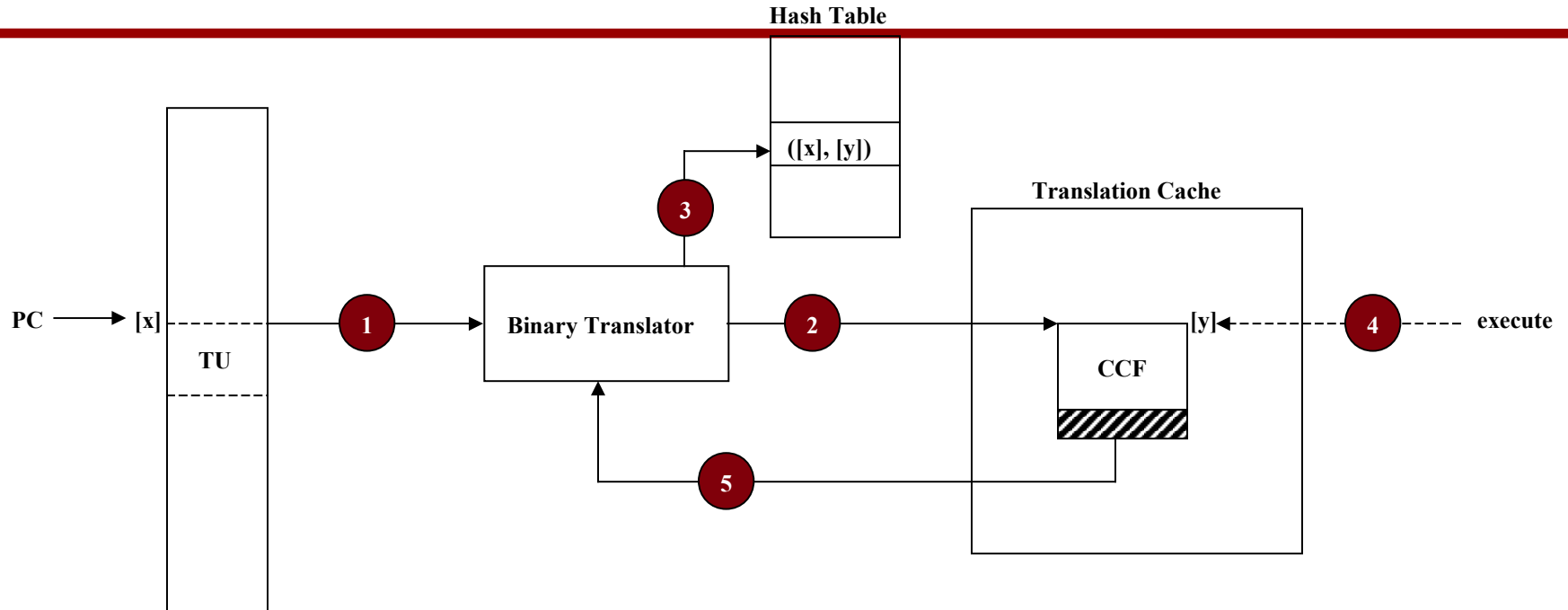
System level – makes no assumption about guest code

Subsetting – translates from full instruction set to safe subset

Adaptive – adjust code based on guest behavior to achieve efficiency




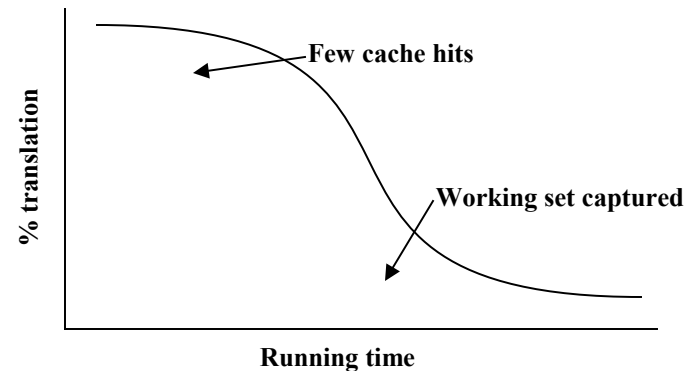
# Binary Translation



TU: translation unit (usually a basic block)

CCF: compiled code fragment

: continuation



# Eliminating faults/traps

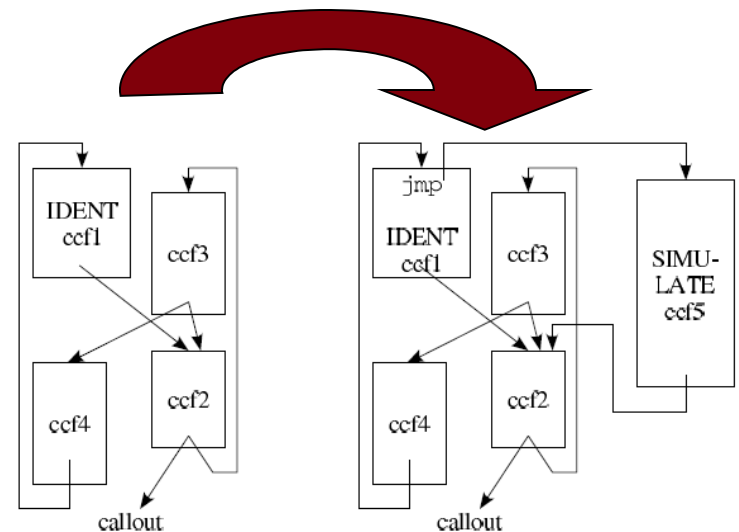
Expensive traps/faults can be avoided

Example: Pentium privileged instruction (rdtsc)

- Trap-and-emulate: 2030 cycles
- Callout-and-emulate: 1254 cycles
- In-TC emulation: 216 cycles

## Process

- Privileged instructions – eliminated by simple binary translation (BT)
- Non-privileged instructions – eliminated by adaptive BT
  - (a) detect a CCF containing an instruction that trap frequently
  - (b) generate a new translation of the CCF to avoid the trap (perhaps inserting a call-out to an interpreter), and patch the original translation to execute the new translation





# Binary Translation Process

**Input: BB**

55 ff 33 c7 03 ...



**Output: CCF**

55 ff 33 c7 03 ...

Each Translator Invocation

- Consume a basic block (BB)
- Produce a compiled code fragment (CCF)

Store CCF in Translation Cache

- Future reuse
- Capture working set of guest kernel
- Amortize translation costs
- Not “patching in place”

At most 12 instructions

BB= sequence of instructions such that if one is executed all the other are executed

Fundamental notion for code optimization

The program code can be represented as a graph where nodes=bb

# Binary Translation Process

```
80304a69 push %ebp
80403a6a push (%ebx)
80403a6c mov (%ebx), ffffffff
80403a72 mov %edx, %esp
80403a74 mov %esp, 81c(%ebx)
80403a7a push %edx
80403a7b mov %ebp, %eax
80403a7d call 80460ba4
```

BB



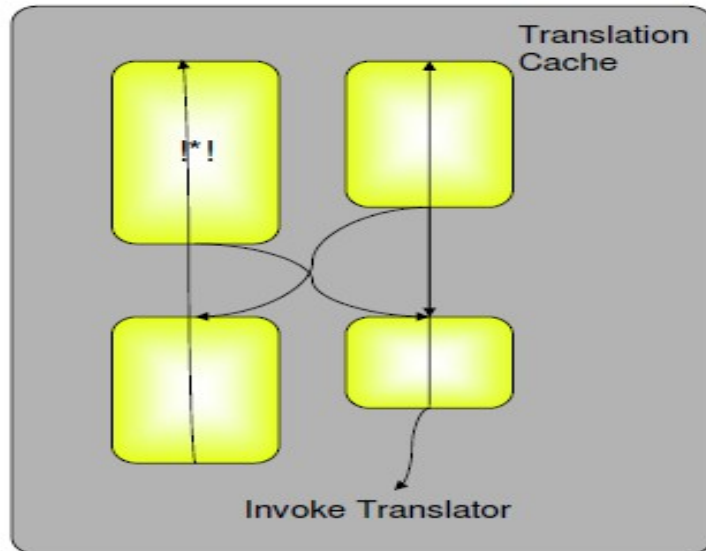
```
25555b0 push %ebp
25555b1 push (%ebx)
25555b3 mov (%ebx), ffffffff
25555b9 mov %edx, %esp
25555bb mov %esp, 81c(%ebx)
25555c1 push %edx
25555c2 mov %ebp, %eax
25555c4 push 80403a82
25555c9 int 3a
25555cb data: 80460ba4
```

CCF

25555c4: push return address

25555c9: invoke translator on callee

# Adaptive Binary Translation Process



## Translated Code Is Fast

- Mostly IDENT translations
- Runs "at speed"

## Except Writes to Traced Memory

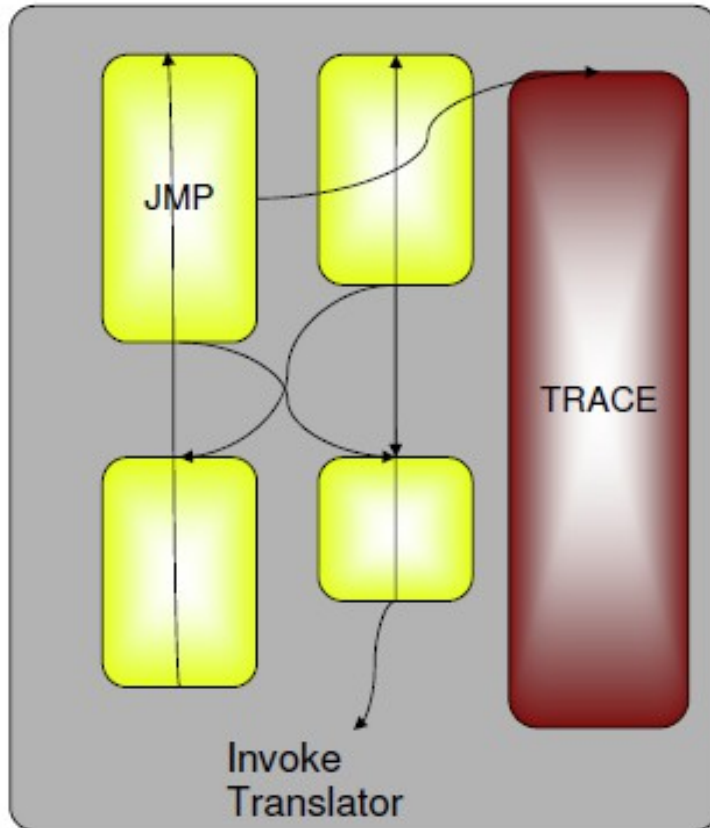
- Page fault (shown as !\*)
- Decode and interpret instruction
- Fire trace callbacks
- Resume execution
- Can take 1000's of cycles

Cache tables are protected  
= trace memory

Detect instructions that trap frequently  
Adapt the translation of these instructions =  
Re-translate to avoid trapping.  
Jump directly to translation.

Adaptive Binary Translation tries to eliminate more and more traps over time.

# Adaptive Binary Translation Process



Detect and Track Trace Faults

Splice in TRACE Translation

- Execute memory access in software
- Avoid page fault
- No re-decoding
- Faster resumption

Faster Traces

- 10x performance improvement
- Adapts to runtime behavior



# Trace Memory

---

## Shadow Page Table

- Derived from primary page table in guest
- VMM must keep primary and shadow coherent

## Trace = Coherency Mechanism

- Write-protect primary page table
- Trap guest writes to primary
- Update or invalidate corresponding shadow
- Transparent to guest



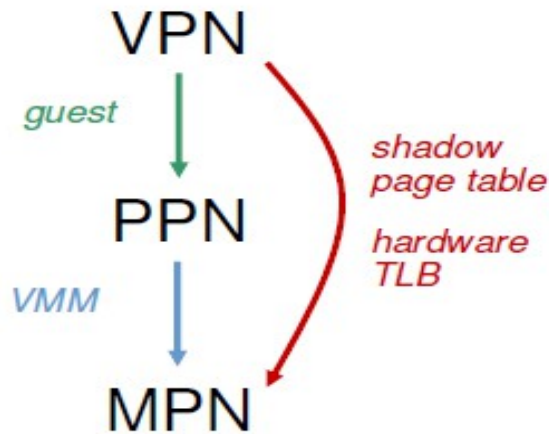
# Shadow Page Table

---

- Shadow page tables are used by the hypervisor to keep track of the state in which the guest "thinks" its page tables should be.
- The guest can't be allowed access to the hardware page tables because then it would essentially have control of the machine
- The hypervisor keeps
  - the "real" mappings guest virtual → host physical in the hardware when the relevant guest is executing
  - a representation of the page tables that the guest thinks it's using "in the shadows," they are not used by the hardware



# Shadow structures



## Traditional VMM Approach

### Extra Level of Indirection

- Virtual → “Physical”  
Guest maps VPN to PPN using primary page tables
- “Physical” → Machine  
VMM maps PPN to MPN

Built incrementally  
at hidden faults

### Shadow Page Table

- Composite of two mappings
- For ordinary memory references  
Hardware maps VPN to MPN
- Cached by physical TLB

A shadow structure records the state of the emulated machine

VPN= virtual page number,

PPN=physical page number

MPN= machine page number

True fault = faults in the emulated machine

Hidden fault = due to the shadow page table



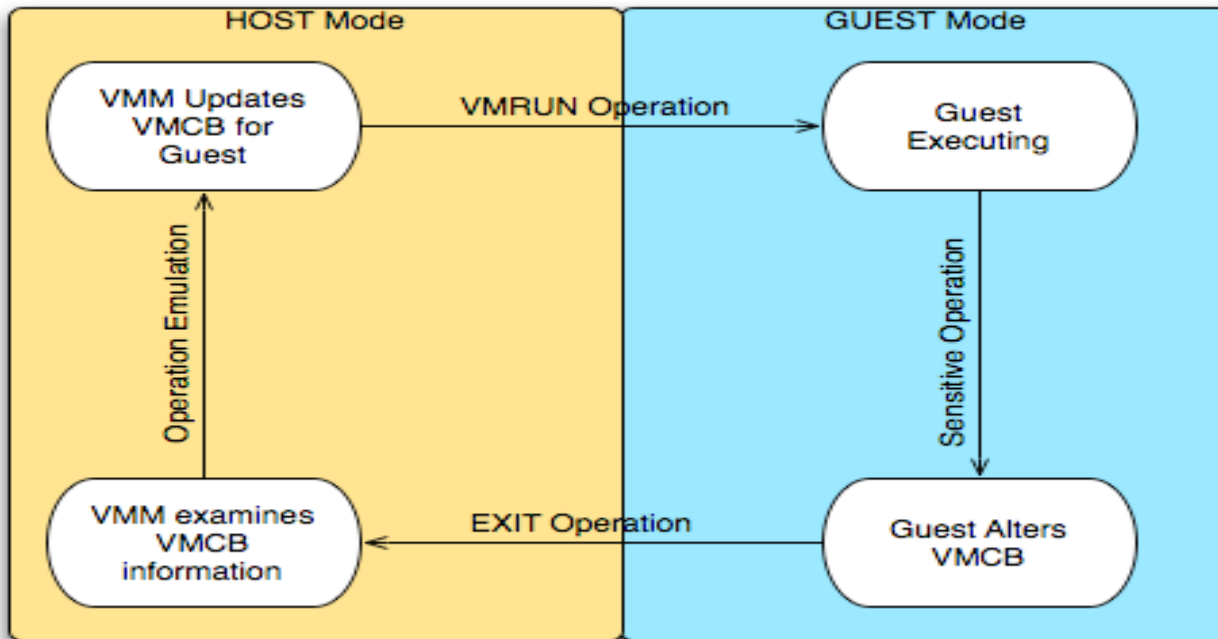
# VMM memory management

---

- VMMs tend to have simple hardware memory allocation policies
  - Static: VM gets 512 MB of hardware memory for life
  - No dynamic adjustment based on load because OSes not designed to handle changes in physical memory...
  - No swapping to disk
- Balloon driver runs inside OS to consume hardware page:

“ESX Server controls a balloon module running within the guest directing it to allocate guest pages and pin them in “physical” memory. The machine pages backing this memory can then be reclaimed by ESX Server. Inflating the balloon increases memory pressure, forcing the guest OS to invoke its own memory management algorithms. Deflating the balloon decreases pressure, freeing guest memory.”
- Identify identical physical pages (e.g., all zeroes) and map those pages copy-on-write across VMs

# x86 Architecture Extensions



Four rings  
For each mode

- VM Exit -> trap to hypervisor (enter host mode)
- VM run -> run the guest OS (enter guest mode)



# Virtual Machine Control Structure (VMCS)

---

1. Data structure to manages VM entries and VM exits.
2. VMCS is logically divided into:
  1. Guest-state area=info on the VM CPU
  2. Host-state area.
  3. VM-execution control fields
  4. VM-exit control fields
  5. VM-entry control fields
  6. VM-exit information fields
3. VM entries = load processor state from the guest-state area.
4. VM exits =
  - 1) save processor state to the guest-state area and the exit reason,
  - 2) load processor state from the host-state area.



# VT-x New instructions

---

VMXON, VMXOFF : To enter and exit VMX-root mode.

VMLAUNCH : initial transition from VMM to Guest , Enters VMX non-root operation mode

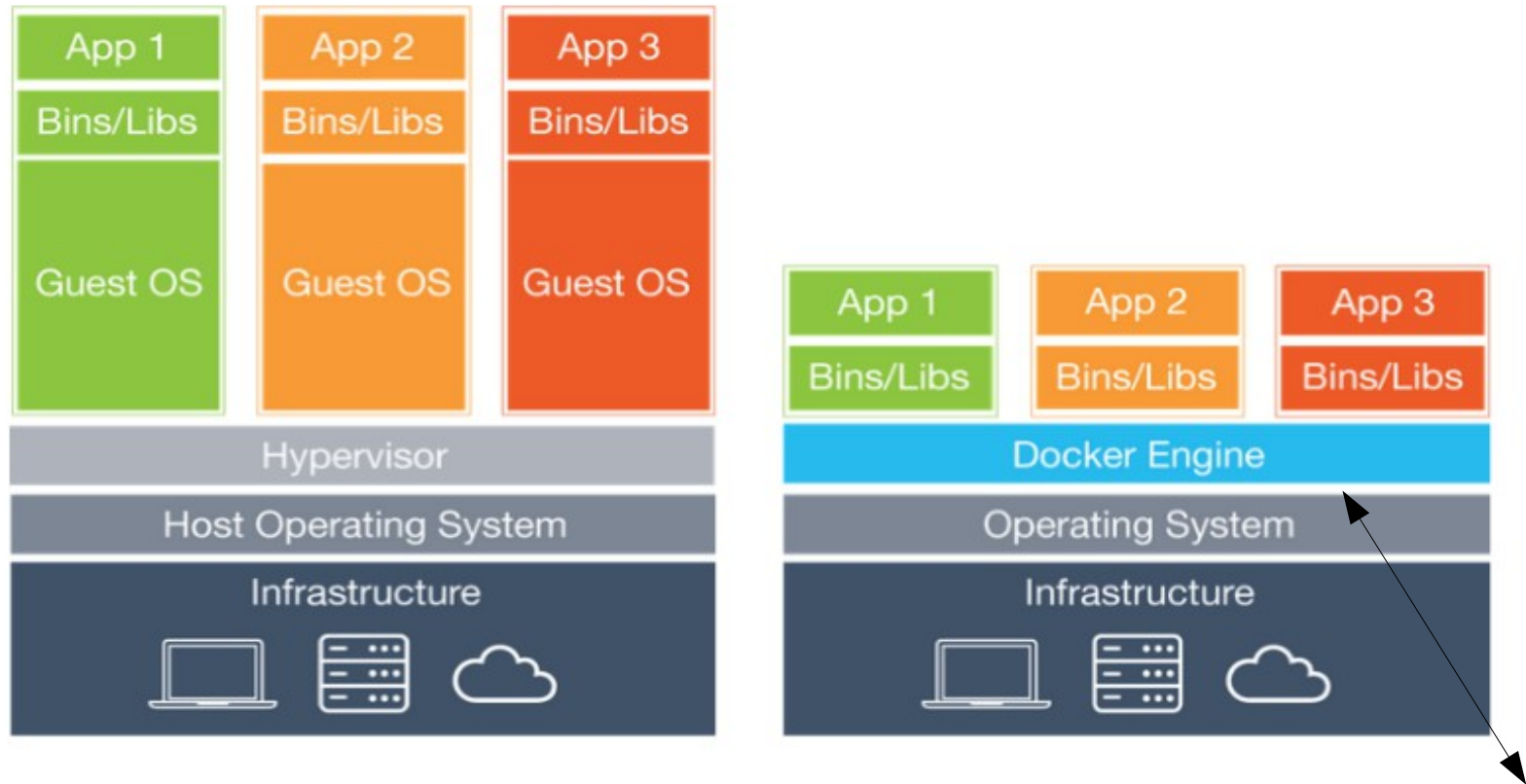
VMRESUME : Used on subsequent entries  
Enters VMX non-root operation mode  
Loads Guest state and Exit criteria from VMCS

VMEXIT : Used on transition from Guest to VMM  
Enters VMX root operation mode  
Saves Guest state in VMCS  
Loads VMM state from VMCS

VMPTRST, VMPTRL: Read and Write the VMCS pointer.

VMREAD, VMWRITE, VMCLEAR : Read from, Write to and clear a VMCS.

# VM vs Container



a lightweight operating system optimized to run Linux containers to reduce the attack surface by minimizing the host environment



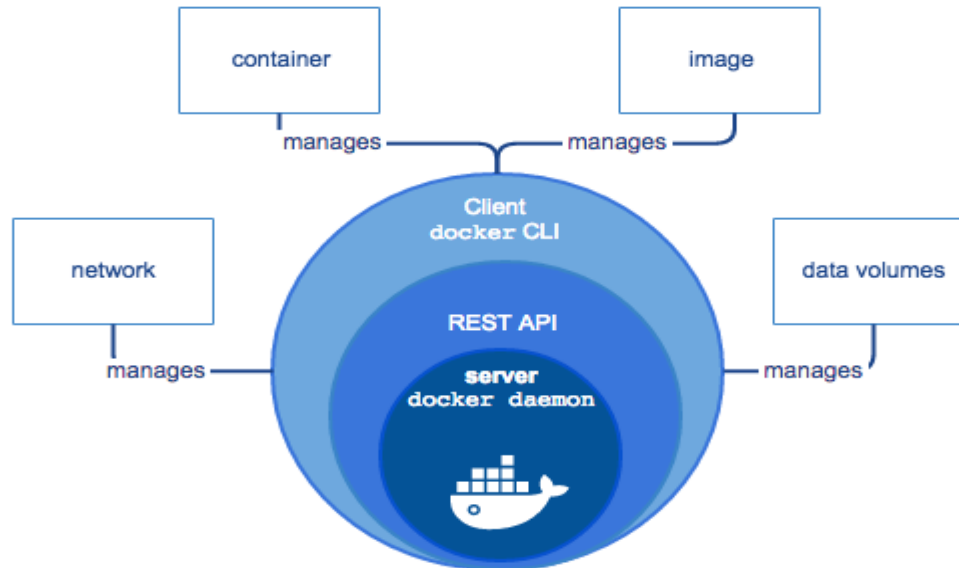
# VM vs Container

---

- **Footprint:** VMs are inherently heavyweights. They need to run a complete OS to be able to run a packaged application. This is needed because the system calls made by the Apps are made to the underlying Guest OS. The Guest OS sends the system calls to the Host OS, via Hypervisor, and then relays the return value of the call back to the App. In the case of containers, the Docker engine does not need a Host OS. All System calls are intercepted by the Docker Engine and are relayed back to the Host OS. Hence, the Docker containers are extremely lightweight.
- **Resources:** VMs should be allocated a defined amount of resources, which cannot be shared between multiple VMs. If you share  $X$  amount of RAM to a particular VM, then this  $X$  amount of RAM would be *dedicatedly* allocated to the VM. In the case of containers though, they utilize the resources as per the need. If a container is running a very lightweight application, it will utilize just the right amount of RAM
- **Automation:** It is possible to create Docker containers on the fly by writing a couple of lines of configuration. Hence, a Docker can be easily integrated with your CI/CD or Deployment tools (like Jenkins, etc.). The same cannot be said about the VMs.
- **Instantiation:** Instantiating a VM instance is a time taking process, sometimes taking tens of minutes. But Docker containers can be started within seconds.
- **Collaboration:** It is very easy to share your Docker images (and containers) with other users. A Docker provides you with Registries which can store and share your images, publicly or privately. VMs are not that easy to share.

# VM vs Container: Docker Engine

- A server which is a type of a long-running program called a Daemon process.
- A REST API which specifies interfaces that programs can use to talk to the Daemon and instruct it what to do.
- A command line interface (CLI) client.







# VM vs Container

---

A Docker uses a client-server architecture. The Docker client talks to the Docker Daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and the Daemon can run on the same system, or you can connect a Docker client to a remote Docker Daemon. The Docker client and Daemon communicate using a REST API, over UNIX sockets or a network interface.

- **Docker Images:** A Docker image is a read-only template with instructions for creating a Docker container. For example, an image might contain an Ubuntu operating system with an Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others. An image may be based on or may extend, one or more of the other images. A Docker image is described in a text file known as a Docker file, which has a simple, well-defined syntax. For more details about images, see the section: *How does a Docker image work?*

Docker images are the build component of Dockers.

- **Docker Container:** A Docker container is a runnable instance of a Docker image. You can run, start, stop, move, or delete a container using the Docker API or CLI commands. When you run a container, you can provide configuration metadata such as networking information or environment variables. Each container is an isolated and secure application platform but can be given access to resources running on a different host or container, as well as persistent storages or databases.



# Container Security

---

- Containers are Linux processes with isolation and resource confinement to run sandboxed applications on a shared host kernel. Your approach to securing containers should be the same as your approach to securing any running process on Linux. Dropping privileges is important and still the best practice. Even better is to create containers with the least privilege possible.
- Containers should run as user, not root.
- Five of the security features available for securing containers :
  - Linux namespaces,
  - Security-Enhanced Linux (SELinux),
  - cgroups,
  - Capabilities
  - secure computing mode (seccomp)



# Container Security

---

- Linux namespaces provide the fundamentals of container isolation. A namespace makes it appear to the processes within the namespace that they have their own instance of global resources. Namespaces provide the abstraction that gives the impression of your own operating system when you are inside a container.
  - SELinux provides an additional security layer to isolate containers from each other and from the host. SELinux can enforce MAC for every user, application, process, and file. SELinux will stop you if you manage to break out of (accidentally or on purpose) the namespace abstraction.
  - Cgroups (control groups) limit, account for, and isolate the resource usage of a collection of processes. Cgroups ensure a container will not be stomped on by another one on the same host and control pseudodevices—an attack vector.
  - Linux capabilities can lock down root in a container. They are distinct privilege units that can be independently enabled or disabled. In containers, you can drop multiple capabilities without impacting the majority of containerized applications.
  - A seccomp profile can be associated with a container to restrict available system calls.
-



# Container vs VM from Cybok

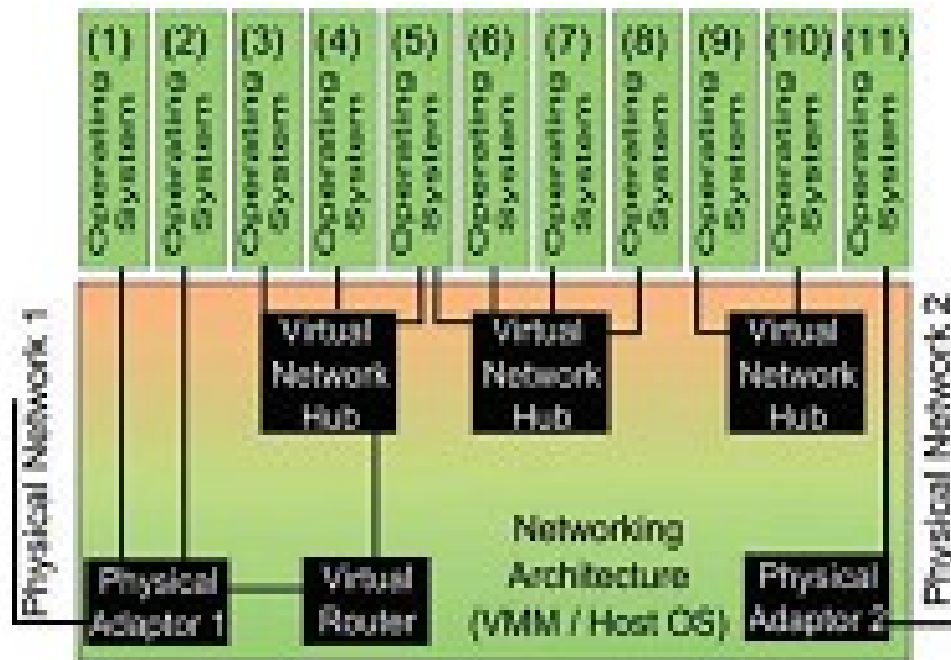
---

Containers are isolated from each other as much as possible (and have their own kernel name spaces, resource limits etc.), but ultimately share the underlying operating system kernel, and often binaries and libraries. Compared to virtual machines, containers are much more lightweight.

However, virtual machines are often perceived as more secure than containers, as they partition resources quite strictly and share only the hypervisor as a thin layer between the hardware and the software.

On the other hand, some people believe that containers are more secure than virtual machines, because they are so lightweight that we can break applications into ‘microservices’ with well-defined interfaces in containers. Moreover, having fewer things to keep secure reduces the attack surface overall.

# A typical virtual architecture



9+10 = isolated network

3-8 = connected network

Ten VMs connected to virtual networks in various arrangements.



# VM detection

---

How a user can detect that the application is running on a VM as a first step to attack the VM itself?

- Detect VM Artifacts in Processes, File System, and/or Registry
- Look for VME Artifacts in Memory
  - The Red Pill (Matrix)
- Look for VME-specific virtual hardware
- Look for VME-specific processor instructions and capabilities



# VM detection – Artifacts in Process etc.

---

Some VMEs insert elements into the Guest that can be easily found

- Running processes or services
- Files and/or directories
- Specific registry keys
  - Some Phatbot malware specimens use this technique
- In a VMware Workstation WinXP Guest:
  - Running “VMtools” service
  - Over 50 different references in the file system to “VMware” and vmx
  - Over 300 references in the Registry to “VMware”
- This method is of limited utility, easily fooled
  - Rootkits tweak the operating system to hide artifacts from users
  - Similar techniques could be applied to hide VME



# VM detection – Artifacts in Memory

---

The Guest system memory map has some differences from the Host memory map

- Strings found in memory
  - By dumping RAM of VMware Workstation WinXP Guest
  - Over 1,500 references to “VMware” in memory
  - Rather a heavy-weight approach, but could be refined to focus on specific regions
- Some critical operating system structures located in different places
- Much quicker, easier, and hard to fool without redesign of VME

One particular memory difference is the location of the Interrupt Descriptor Table (IDT)

- On Host machines = it is typically low in memory
- On Guest machines = it is typically higher in memory
- Cannot be the same, because the processor has a register pointing to it (IDTR)

Memory technique is usable across different VMEs (VirtualPC and VMware) and more difficult to fool





# VM detection – Memory – Blue Pill - 1

---

- In November 2004, Joanna Rutkowska released a tool, "The Red Pill" that reliably detects virtual machine usage without looking for file system artifacts  
<http://invisiblethings.org/index.html#redpill>
- This tool runs a single machine language instruction, SIDT  
“Store Interrupt Descriptor Table”
- This instruction, which can be run in user mode, takes the location of the Interrupt Descriptor Table Register (IDTR) and stores it in memory where it is analyzed



## VM detection – Memory – Blue Pill - - 2

---

- On VMware guest machines, the IDT is typically located at 0xffXXXXXX
- On VirtualPC guests, it is located at 0xe8XXXXXX
- On host operating systems, it is located lower than that,
  - 0x80ffffff (Windows)
  - 0xc0ffffff (Linux)
- The Red Pill merely looks at the first byte returned by SIDT
  - If it's greater than 0xd0, you've got a virtual machine
  - If it is less than or equal to 0xd0, you are in a real machine



# Look for VME-Specific Virtual Hardware

---

- VME introduces virtualized hardware
  - Network
  - USB controller
  - Audio adapters
- Some of these have distinct fingerprints
  - MAC addresses on NICs
  - USB controller type
  - SCSI device type
- Also, anomalies in the way the Guest system clock is updated
- Easy-to-write code, but likely easily fooled



# Look for VME-Specific Virtual Hardware

---

## Linux version of Doo

- Simple shell script looks for “VMware” located in:
  - /proc/iomem
  - /proc/ioports
  - /proc/scsi/scsi
  - dmesg command (print kernel ring buffer; holds boot messages and related logs from kernel)
- Also looks in dmesg output for “BusLogic BT-958” and “pcnet32” - These are known VMware devices

## Windows version of Doo:

- Uses Windows Scripting Host to read 2 registry keys associated with SCSI to look for “VMware”
- Uses WSH to read 2 other registry keys associated with specific class ID of VMware virtualized hardware



# Look for VME-Specific Processor Instructions and Capabilities

---

Some VMEs introduce “extra” machine-language instructions beyond the standard x86 instruction set to foster Guest-to-Host communication or for other virtualization issues

- Code could play a non-standard x86 instruction used by VMware, VirtualPC, or Xen into processor to see if it rejects it or handles it
- Alternatively, code could look for unusual processor behavior associated with “normal” machine language instructions



# Look for VME-Specific Processor Instructions and Capabilities

---

To detect VirtualPC, VMDetect:

- Registers its own handler for invalid OpCodes
- Runs a VirtualPC-specific non-standard IA32 instruction
- If the processor runs the instruction, it is VirtualPC
- If the handler for invalid OpCodes is called, it it's a real machine



# VMware Detection with VMDetect at the Machine Language Level

---

- The machine language looks for the VMware guest-to-host channel, by checking for a strange processor property of VMware guests.
- This code attempts to invoke the VMware guest-to-host communication channel created by overloading the functionality of a specific x86 instruction, “IN.”
- The IN instruction is used to read a byte, word, or dword of data from an I/O port. It has two parameters:
  - the register that is the data destination and
  - the port is to be accessed. This number is placed in the processor register DX before the instruction is executed.
- VMware monitors any use of the IN instruction, and captures any I/O destined for a specific port number (0x5658) but only when the value of another processor register, EAX, is a very specific “magic” number.



# VMware Detection with VMDetect at the Machine Language Level

---

- The code detects VMware by first loading EAX with this magic value, “VMXh”. A specific “command” for VMware to process (in this case 0x0A or decimal 10) is loaded in ECX and parameter data (in this case, 0) is loaded into register EBX. The special port number (0x5658 which also stands for the characters “VX”) is loaded into register EDX. Then the code executes IN.
- On a non-virtual machine, this will cause a processor exception and trigger specifically provided exception handling code within the software.
- On a VMware machine, the instruction will be monitored and allowed to succeed without error by VMware which will then change the values in the processor’s registers before returning execution to the code.
- The end result will be that the magic value, “VMXh” will be moved into register EBX. The code then compares EBX to this value and continues on, knowing for certain that it is running in a virtual environment.





# VMware Detection with VMDetect at the Machine Language Level

---

```
MOV EBX,0
MOV ECX,0A
MOV EDX,5658 <-- "VX"
IN EAX,DX <-- Check for VMWare
CMP EBX,564D5868
```

- First, the EAX register is loaded with the “magic value” to the use the communication channel between the real and the virtual machine(“VMXh”)
  - ECX is loaded with a command value (0x0A which is used to request VMware version information from the host)
  - Any parameters needed for the command (in this case there are none) are loaded in EBX
  - Finally, the IN instruction (used for port I/O) is used, which would normally attempt to load data from port 0x5658 (“VX”)
  - If we are outside VMware, a privilege error occurs
  - If we’re inside VMware, the magic value (VMXh) is moved to register EBX; otherwise it is left at 0
  - Based on the version values returned, we can determine the VMware product
-



# Compatibility vs Transparency

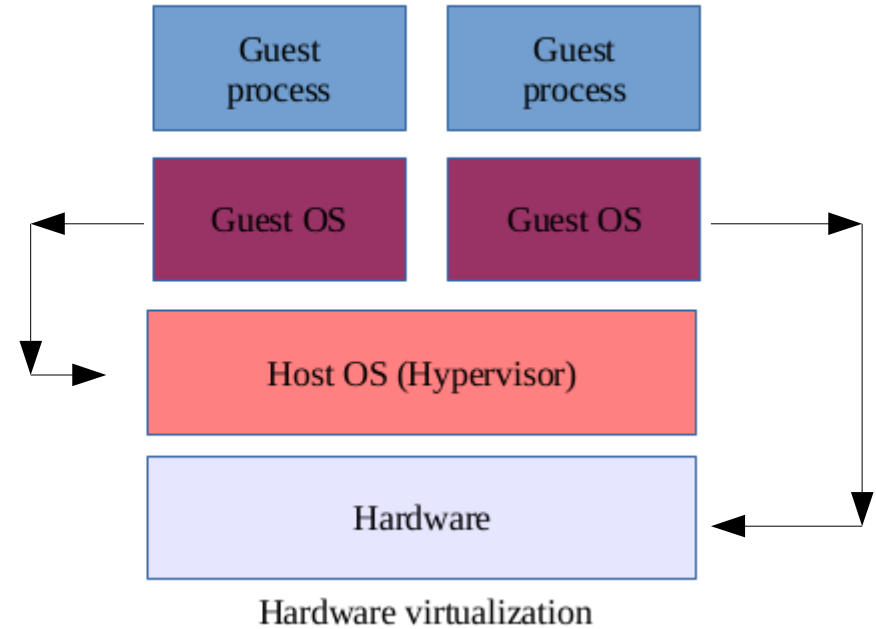
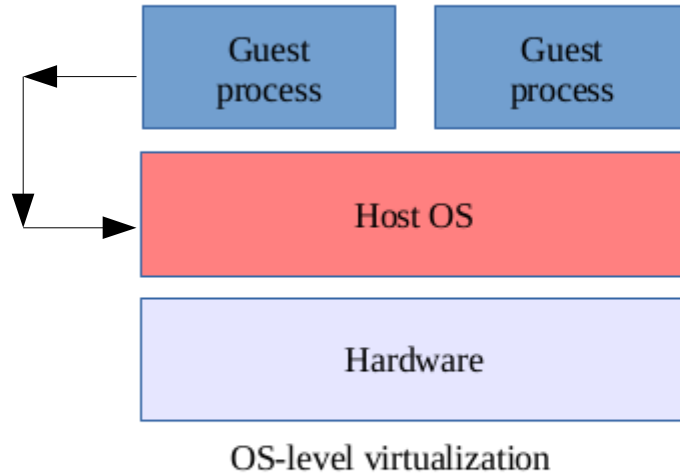
---

- Compatibility is Not Transparency: VMM Detection Myths and Realities, HotOS-XI, May 2007

the goal of VM technology has been, till now, a minimal loss of performance due to the virtualization

- “The belief that VMM transparency is possible is based on a mistaken intuition that compatibility and performance imply transparency i.e. that once VMMs are able to run software for native hardware at native speeds, they can be made to look like native hardware under close inspection. This is simply not the case”
  - Why bother about transparency?
  - Compatibility is not isolation and is not security
-

# VM vulns





# VM vulns: CVE-2007-4496

---

## VMWare ESX 3.0.1

- <http://www.vmware.com/support/vi3/doc/esx-8258730-patch.html>
- Found by Rafal Wojtczuk (McAfee)
- September 2007
- Guest OS can cause memory corruption on the host and *potentially* allow for arbitrary code execution on the host



# VM vulns: CVE-2007-0948

---

## Microsoft Virtual Server 2005 R2

- <http://www.microsoft.com/technet/security/bulletin/ms07049.msp>
- Found by Rafal Wojtczuk (McAfee)
- August 2007
- Heap-based buffer overflow allows guest OS to execute arbitrary code on the host OS



# CVE-2007-4993

---

## Xen 3.0.3

- [http://bugzilla.xensource.com/bugzilla/show\\_bug.cgi?id=1068](http://bugzilla.xensource.com/bugzilla/show_bug.cgi?id=1068)
- Found by Joris van Rantwijk
- September 2007
- By crafting a grub.conf file, the root user in a guest domain can trigger execution of arbitrary Python code in domain 0
- Domain 0 = Xen Hypervisor

**Grub = GRand Unified Bootloader**



# Cloudburst -1

---

Combination of 3/4 bugs in the VMware emulated video device

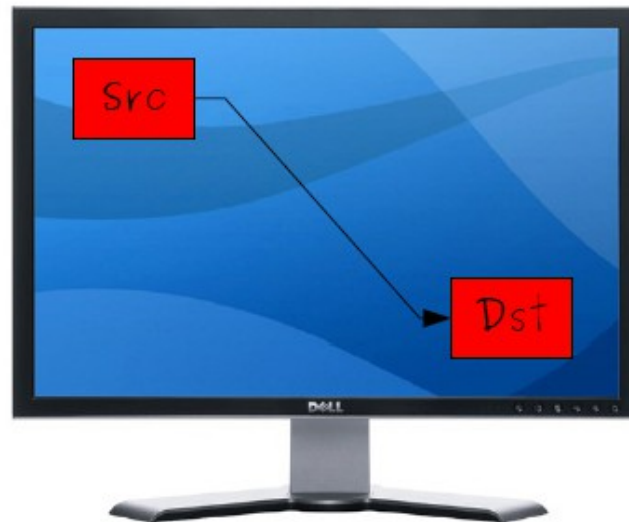
- Host memory leak into the Guest
- Host arbitrary memory write from the Guest
  - Relative
  - Absolute
- And some additional DEP friendly goodness

Reliable Guest to Host escape on recent VMware products

## Cloudburst - 2

### SVGA\_CMD\_RECT\_COPY

- Copies a rectangle in the Frame Buffer from a source X, Y to a destination X, Y

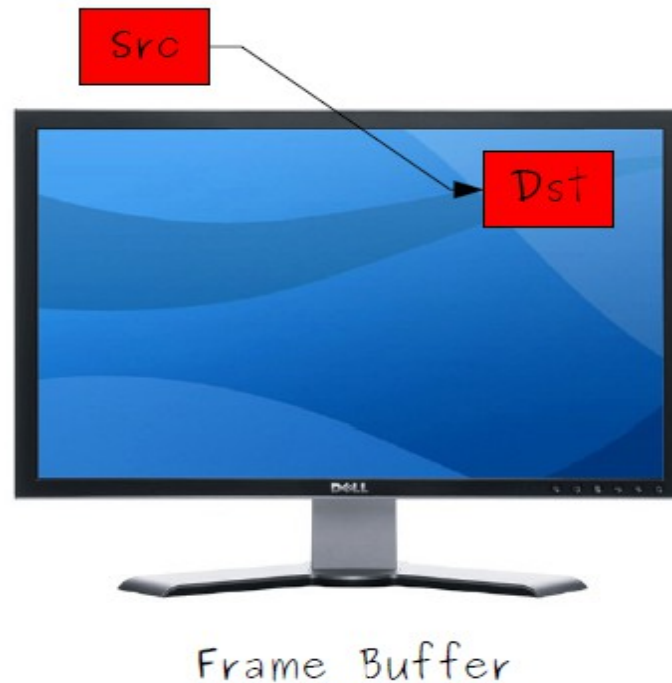


Frame Buffer



## Cloudburst - 3

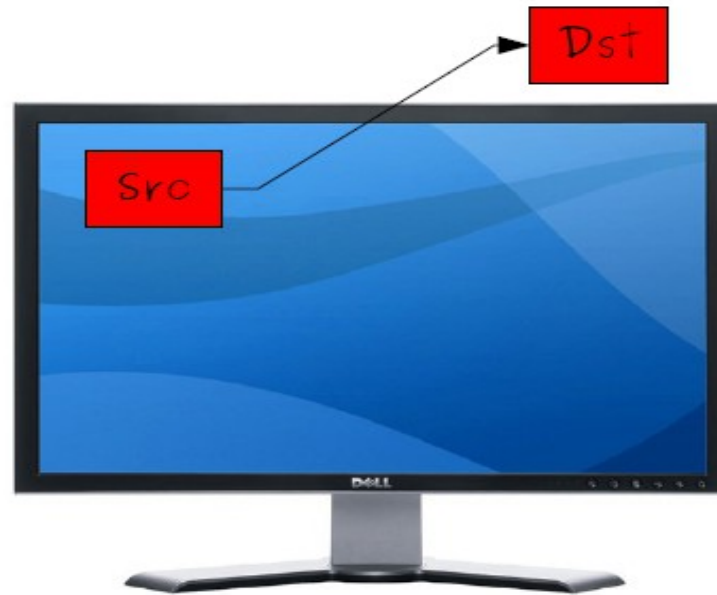
- Boundaries checks on the source location can be bypassed



**Can be used to access information on the host**

## Cloudburst - 4

- Boundaries checks on the destination location can be bypassed (to a lower extent than source)



Frame Buffer

**Can be used to update information on the host**

## Cloudburst - 5

---

### SVGA\_CMD\_DRAW\_GLYPH

- Draws a glyph into the frame buffer
- Requires `svga.yesGlyphs="TRUE"`



Virtual screen

## Cloudburst - 6

### SVGA\_CMD\_DRAW\_GLYPH

- There is no check on the X, Y where the glyph is to be copied



Virtual screen



## Cloudburst - 7

---

- To defeat Vista Data Execution Prevention Technology, the attack has 12 steps
- This confirms that in the case of clouds attacks are possible but they may require a larger number of steps than traditional ones
- Attack graphs can simplify the detection of attacks in this class

And now back to distinct vulnerabilities ....

# Statistics and some general attacks

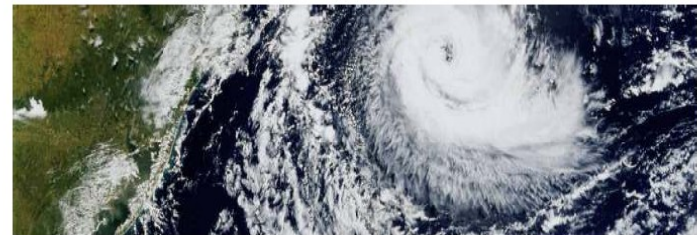
- Virtualization System Security  
Bryan Williams, IBM X-Force  
Advanced Research  
Tom Cross, Manager, IBM X-  
Force Security Strategy
- Describes some statistics  
about vulnerabilities in  
virtualization components
- Discuss a first set of VM  
attacks

IBM

## Virtualization System Security

Bryan Williams, IBM X-Force Advanced Research

Tom Cross, Manager, IBM X-Force Security Strategy





Group	Feature	Benefits	Threats	Vulnerabilities	Attacks	Confidentiality	Integrity	Availability	Non-repudiation	Authenticity
VM	store VM as image	backup VM	VM image modification	software	VMM ► VM	-	-	+		
	modified VM software	security checks	attack VMM	software	VM ► VMM	+	+	+	+	+
VMM	small footprint	fewer vulnerabilities	VMM rootkit	software	VMM ► VM	+	+	+	+	+
	hierarchical control	control untrusted VM	enlarged footprint, VM escape	Software	VM ► VMM	+	+	+	+	+
	isolation between processes	isolate untrusted VM	N/A	covert channels	VM ► VM	+	+	+	+	+
	logging	store log securely	N/A	N/A	N/A		+		+	
	load balancing	prevent DOS	N/A	software	VM ► VM			+		
	copy and backup VMs	facilitate backup	VM branching	management	N/A	-		+		
	introspection	virus scan, attestation	introspection misuse	software	VMM ► VM	±	±	+		
	attestation	authenticate VM	N/A	N/A	N/A	+	+		+	+
	interference	prevent and stop attacks	intervention misuse	software	VMM ► VM	±	±	±	±	±
	power functions	recover from errors	sleeper-exploit	management	VMM ► VM			+		
	networking	isolation	network traffic snoop	software	VMM ► VM, network ► VM	±	±	±	±	±
	rollback	rollback illegal action, recover from errors	rollback patch	management	VMM ► VM	-	-	+	-	-
VM management	facilitate control	abuse	management		±	±	+	±	±	
VMMM	transfer	migrate if error	DOS, in-transfer modification, transfer off-site	management	Network ► VM, VMM ► VM, VMMM ► VMM	-	-	+	-	-
	replication	anti DOS	clone, replicate	management	VMMM ► VMM	-	-	+	-	-
	load balancing	anti DOS	abuse	management	VMMM ► VMM	-	-	+	-	-
	patching	facilitate patching	N/A	N/A	N/A	±	±	±		
	VMM management	facilitate control	abuse	abuse	VMMM ► VMM	±	±	+	±	±
emergent	loss of uniqueness	N/A	exploits	management	N/A	-		+	-	-
	loss of location-boundedness	N/A	exploits	management	N/A	-		+	-	-
	loss of monotonicity	N/A	exploits	management	N/A		-		-	-



# Other classification

## Security Implications of Virtualization: A Literature Study

Andre van Cleeff, Wolter Pieters, Roel Wieringa

- Proposes a classification in terms of the attack enabled by a vulnerability

Threat source	Explanation
Network ► VMMM	An outsider attacks the VMMM
Network ► VMM	An outsider attacks the VMM
Network ► VM	An outsider attacks the VM
VMMM ► VMM	A VMMM attacks a VMM
VMM ► VM	A VMM attacks a VM
VM ► VM	A VM attacks another VM

VMMM = virtual machine monitoring and management

- It points out an important feature of virtualization: loss of monotonicity

Virtualization technology causes a server's history to stop being a straight line. Instead it becomes a graph, where branches are made on replication and copy operations, and a previous state can be reached when a restore is performed. Data cannot be deleted easily, there can be many copies and the VM can be restored to an earlier version.





# Loss of monotonicity

---

- a) log of a virtual machine ???
- b) physical location when several copies of the same machine exist
- c) patching of virtual machine. Reverting to an old version may remove the patch. Several versions may be run
  - One patched
  - One unpatched
- d) if several versions exist, all must be located



# Reincarnation Attack: A bad usage of loss of monotonicity

## Controlled Reincarnation Attack to Subvert Digital Rights Management

F. John Krautheim and Dhananjay S. Phatak

- A licensing mechanism based solely on local state (system time, processor identification number ) is defeated
  - by capturing the state of the VM immediately after activating a license
  - by restarting the VM from the same point at any reactivation
- since the VM can be contained in a file, it can easily be distributed in the pristine state and multiple copies of the software can be used simultaneously
- this attack is easily defeated by using an ongoing communication with an activation server to ensure that the license remains valid throughout the session and life of the product.
- A more complex attack captures the communications between the VM and activation server and performs a replay attack. Since the VM is started in a known state, all communications should be exactly the same every time, so that the malicious middleware fakes the software into believing it is talking to a real activation server.

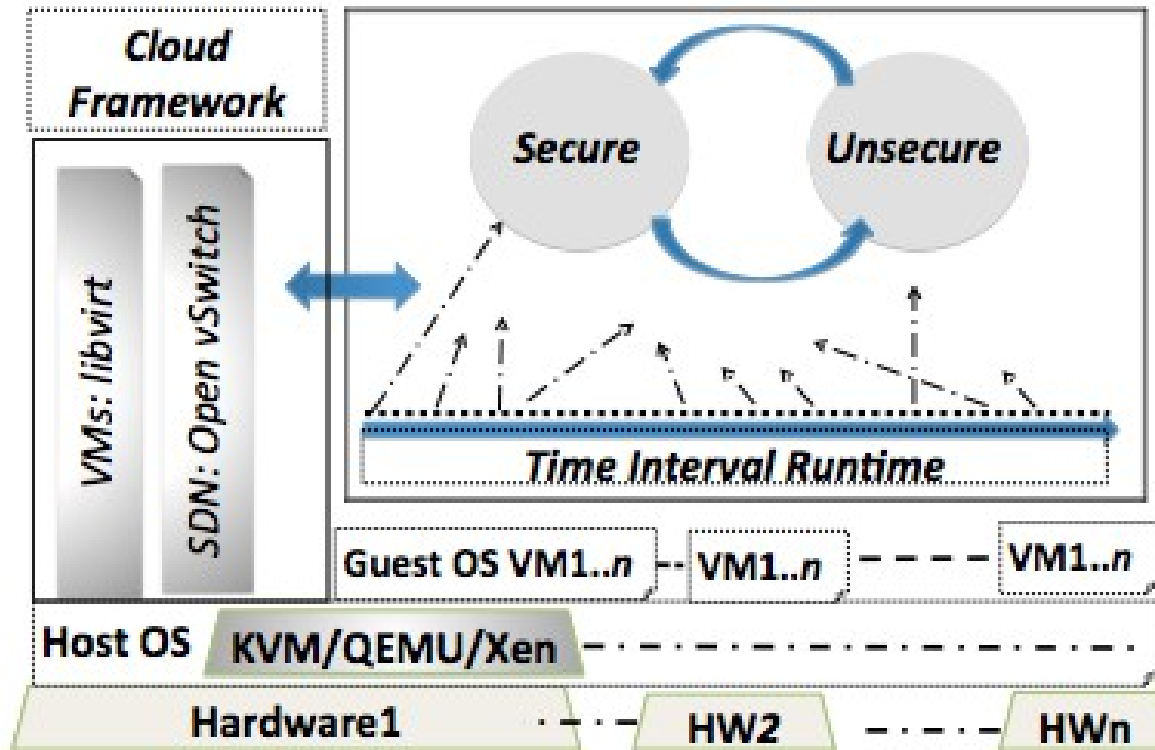


# Virtual Reincarnation : A good usage of loss of monotonicity

---

- The main idea is to stress diversity by running an application on distinct nodes and distinct OSes at distinct times
- A virtual machine is migrated to a distinct node so that if the underlying VMM or hardware has been attacked, the attacker loses control
- In distinct moments the application is run on a distinct VM so that if the OS is attacked, the attacker loses control
  - This can be implemented by transmitting to the second machine a log of the operations of the first one
  - The second machine is frozen when the first one runs and then the first is frozen and the second one is defroze
  - Requires a monitoring infrastructure
  - Activated by intrusion detection

# Overall Architecture

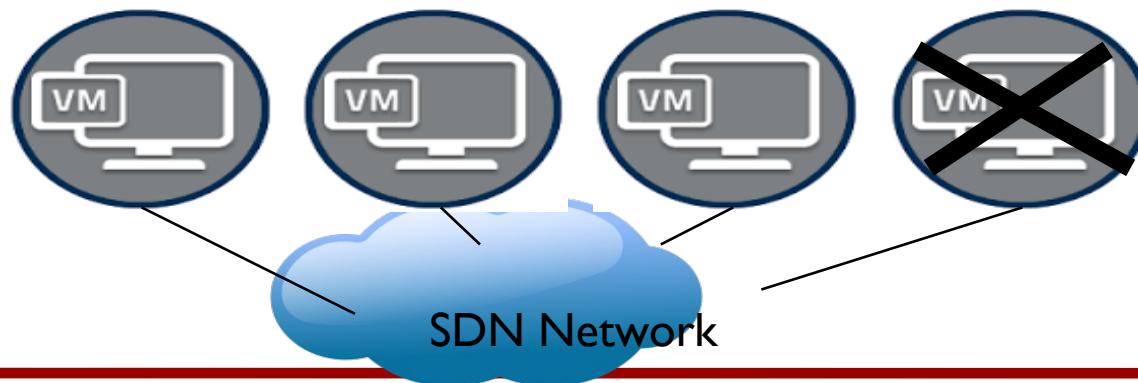


## Components:

- (1) Virtual Reincarnation (ViRA)
- (2) Proactive Monitoring
- (3) SDN Network Dynamics
- (4) Systems States and Application Runtime

## Overall Architecture: Details

- Nodes run a distributed application on a given platform for a controlled period of time
- The running time is chosen in a way that successful ongoing attacks become ineffective
- The new fresh machine will integrate to the system and continue running the application after its data is updated



# Virtual Reincarnation

---

- Randomization and diversification technique where nodes (virtual machines) running a distributed application vanish and reappear on a different virtual state with different guest OS, Host OS, hypervisor, and hardware .

Improve  
Resiliency

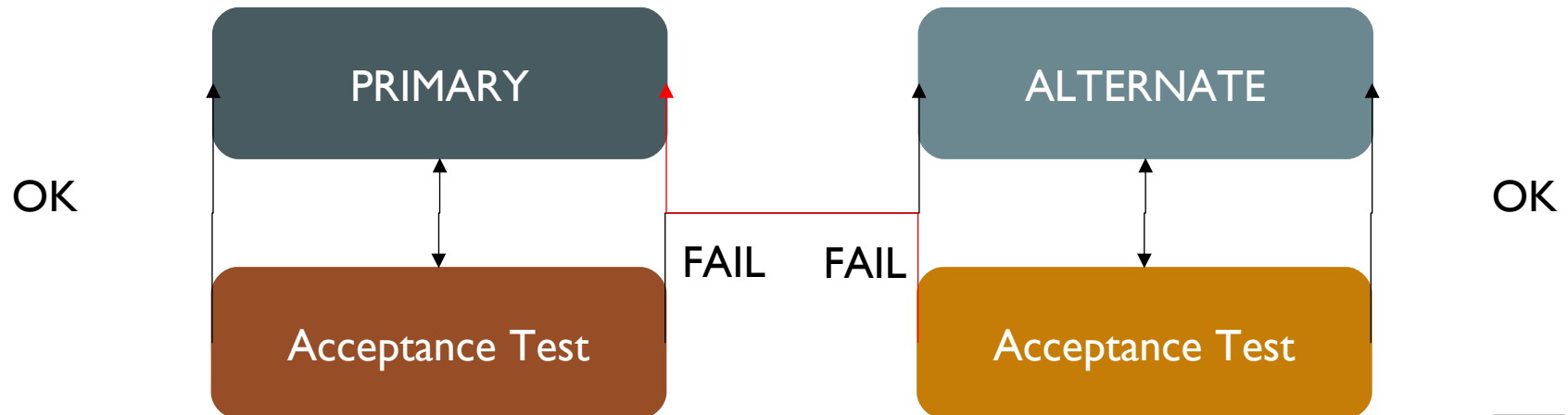
Improve  
Anti-Fragility

Virtualized  
Environment



# How do we create replicas?

- Primary VM runs as no failures are detected.
- Alternate VM takes place when a failure occurs
- Acceptance tests are adjusted independently to guarantee system operation
- Alternate learn from Primary and become more robust to failures/attacks experimented by primary



# How do we create replicas?

- Primary VM runs as no failures are detected.
- Alternate VM takes place when a failure occurs
- Acceptance tests are adjusted independently to guarantee system operation
- Alternate learn from Primary and become more robust to failures/attacks experimented by primary

