CE 874 - Secure Software Systems

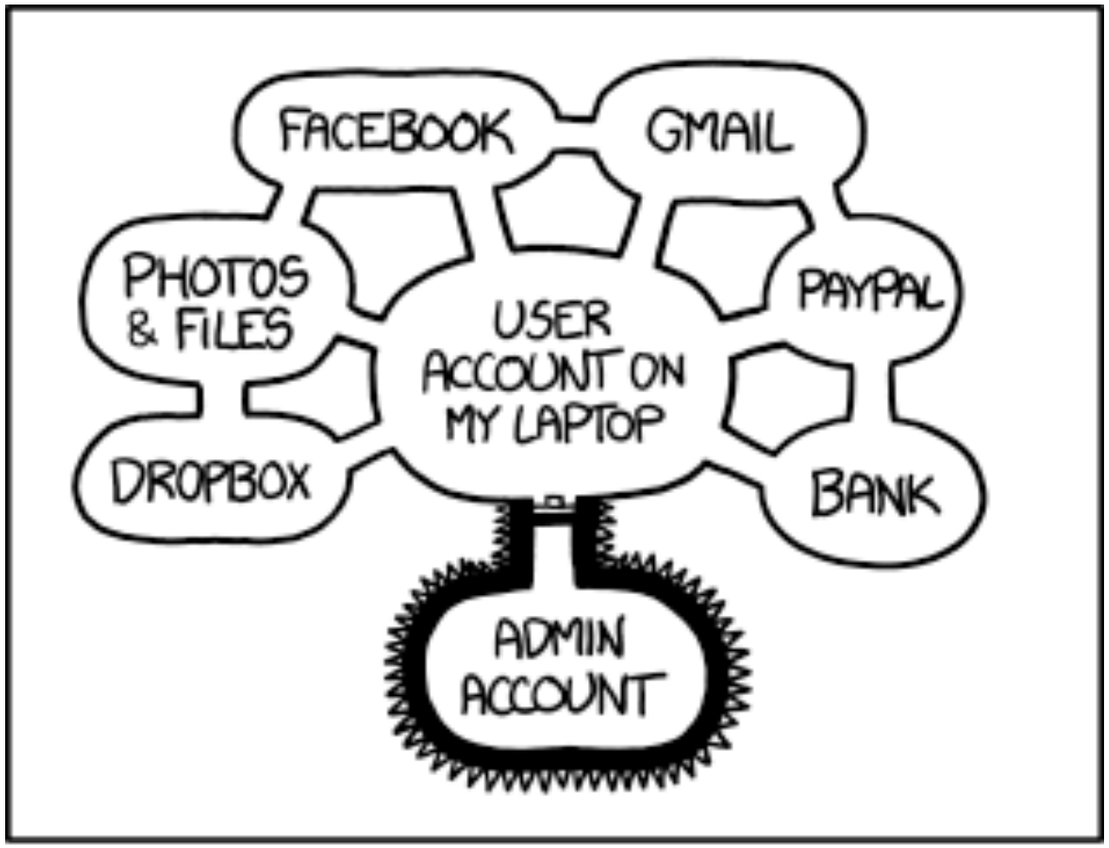
Secure Architecture II

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Secure Architecture

- How to come up with a secure architecture?
- What design principals is should be followed?
- What are the available mechanisms?
- How do you trust the code getting executed?



IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY, AND IMPERSONATE ME TO MY FRIENDS,

BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION. xkcd.com

Example: Virtual Machines

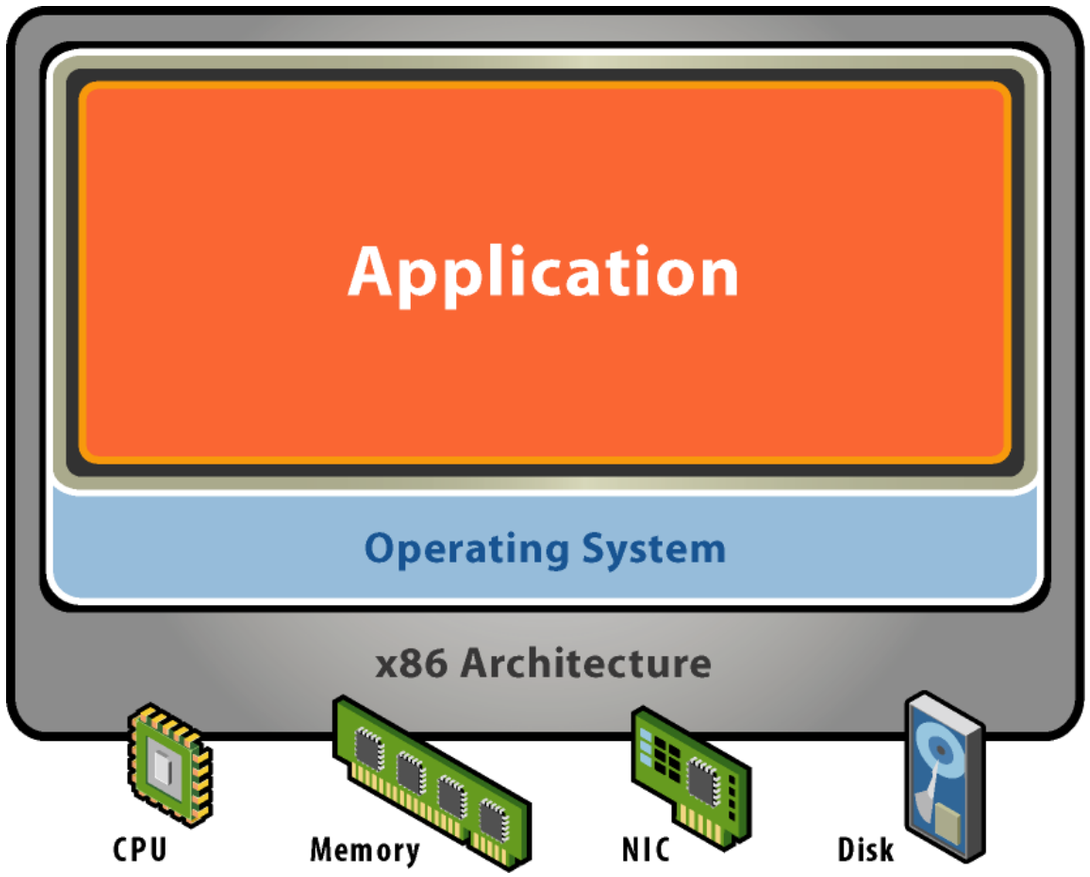
The idea of Virtualization: from 1960's

- IBM VM/370 – A VMM for IBM mainframe
    - Multiple OS environments on expensive hardware
    - Desirable when few machine around
- Popular research idea in 1960s and 1970s
    - Entire conferences on virtual machine monitors
    - Hardware/VMM/OS designed together
    - Allowed multiple users to share a batch oriented system
- Interest died out in the 1980s and 1990s
    - Hardware got more cheaper
    - Operating systems got more powerful (e.g. multi-user)

A Return to Virtual Machines

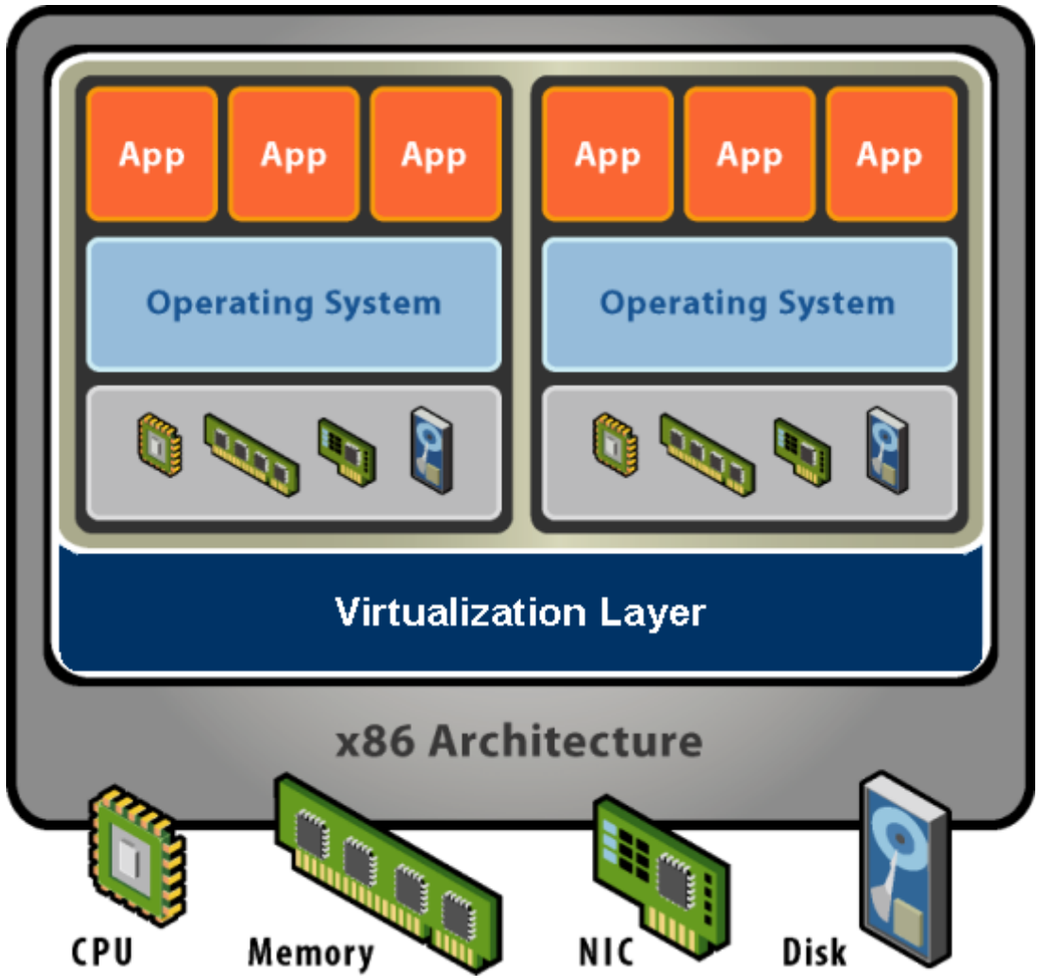
- Disco: Stanford research project (SOSP ’97)
  - Run commodity OSes on scalable multiprocessors
  - Focus on high-end: NUMA, MIPS, IRIX
- Commercial virtual machines for x86 architecture
  - VMware Workstation (now EMC) (1999-)
  - Connectix VirtualPC (now Microsoft)
- Research virtual machines for x86 architecture
  - Xen (SOSP ’03)
  - plex86
- OS-level virtualization
  - FreeBSD Jails, User-mode-linux, UMLinux

Starting Point: A Physical Machine



- Physical Hardware
  - Processors, memory, chipset, I/O devices, etc.
  - Resources often grossly underutilized
- Software
  - Tightly coupled to physical hardware
  - Single active OS instance
  - OS controls hardware

What is a Virtual Machine?



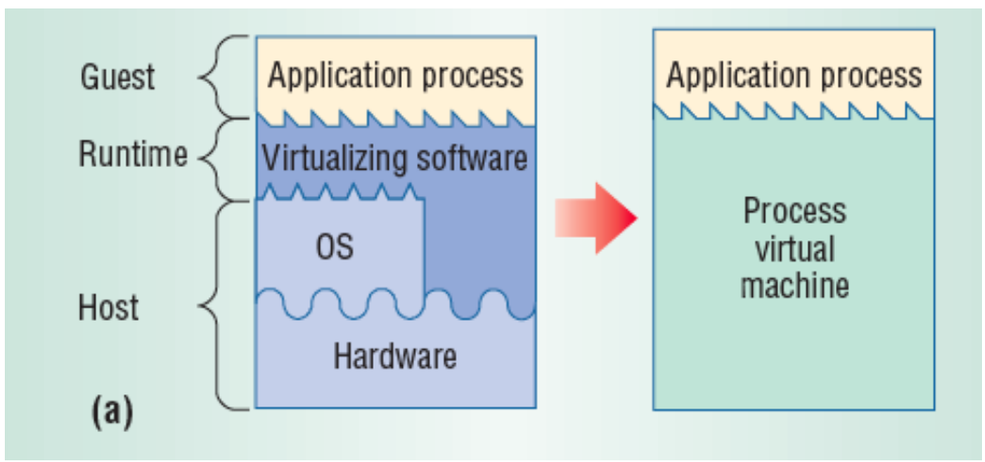
- Software Abstraction
  - Behaves like hardware
  - Encapsulates all OS and application state
- Virtualization Layer
  - Extra level of indirection
  - Decouples hardware, OS
  - Enforces isolation
  - Multiplexes physical hardware across VMs

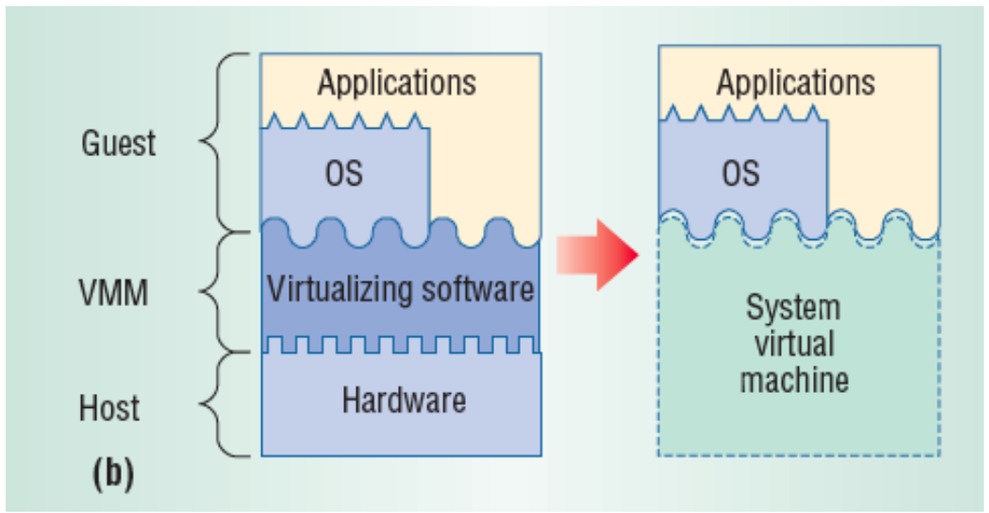
Virtualization Properties, Features

- Isolation
  - Fault isolation and Performance isolation (+ software isolation, …)
- Encapsulation
  - Cleanly capture all VM state and Enables VM snapshots, clones
- Portability
  - Independent of physical hardware
  - Enables migration of live, running VMs (freeze, suspend,…)
  - Clone VMs easily, make copies
- Interposition
  - Transformations on instructions, memory, I/O
  - Enables transparent resource overcommitment, encryption, compression, replication …

Types of Virtualization

- Process Virtualization (Figure [a])
  - Language-level Java, .NET, Smalltalk
  - OS-level processes, Solaris Zones, BSD Jails, Docker Containers
  - Cross-ISA emulation Apple 68K-PPC-x86
- System Virtualization (Figure [b])
  - VMware Workstation, Microsoft VPC, Parallels
  - VMware ESX, Xen, Microsoft Hyper-V





Types of VMs – Emulation

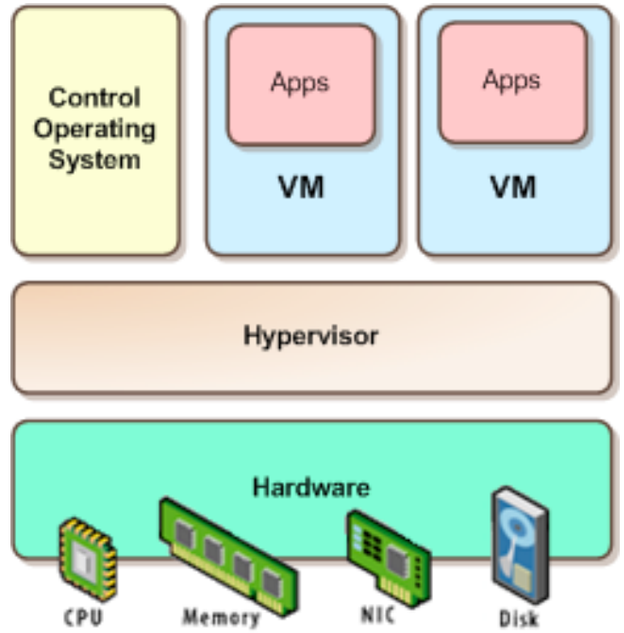
- Another (older) way for running one OS on a different OS
  - Virtualization requires underlying CPU to be same as guest was compiled for while Emulation allows guest to run on different CPU
- Need to translate all guest instructions from guest CPU to native CPU
  - Emulation, not virtualization
- Useful when host and guest have differnet processor architectures
  - Company replacing outdated servers with new servers containing different CPU architecture, but still want to run old applications
- Performance challenge – order of magnitude slower than native code
  - New machines faster than older machines so can reduce slowdown
- Where do you think it is used still?
- Very popular – especially in gaming where old consoles emulated on new

VMs – Application Containers

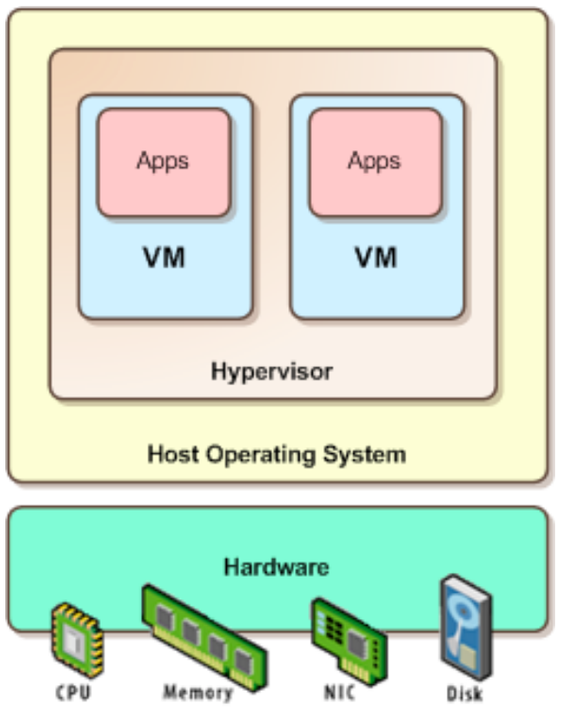
- Some goals of virtualization are segregation of apps, performance and resource management, easy start, stop, move, and management of them
- Can do those things without full-fledged virtualization
  - If applications compiled for the host operating system, don't need full virtualization to meet these goals
- Oracle containers/zones for example create virtual layer between OS and apps
  - Only one kernel running – host OS
  - OS and devices are virtualized, providing resources within zone with impression that they are only processes on system
  - Each zone has its own applications; networking stack, addresses, and ports; user accounts, etc
  - CPU and memory resources divided between zones
    - Zone can have its own scheduler to use those resources

Types of System Virtualization

- Native/Bare metal (Type 1)
  - Higher performance
  - ESX, Xen, HyperV



- Hosted (Type 2)
  - Easier to install
  - Leverage host's device drivers
  - VMware Workstation, Parallels

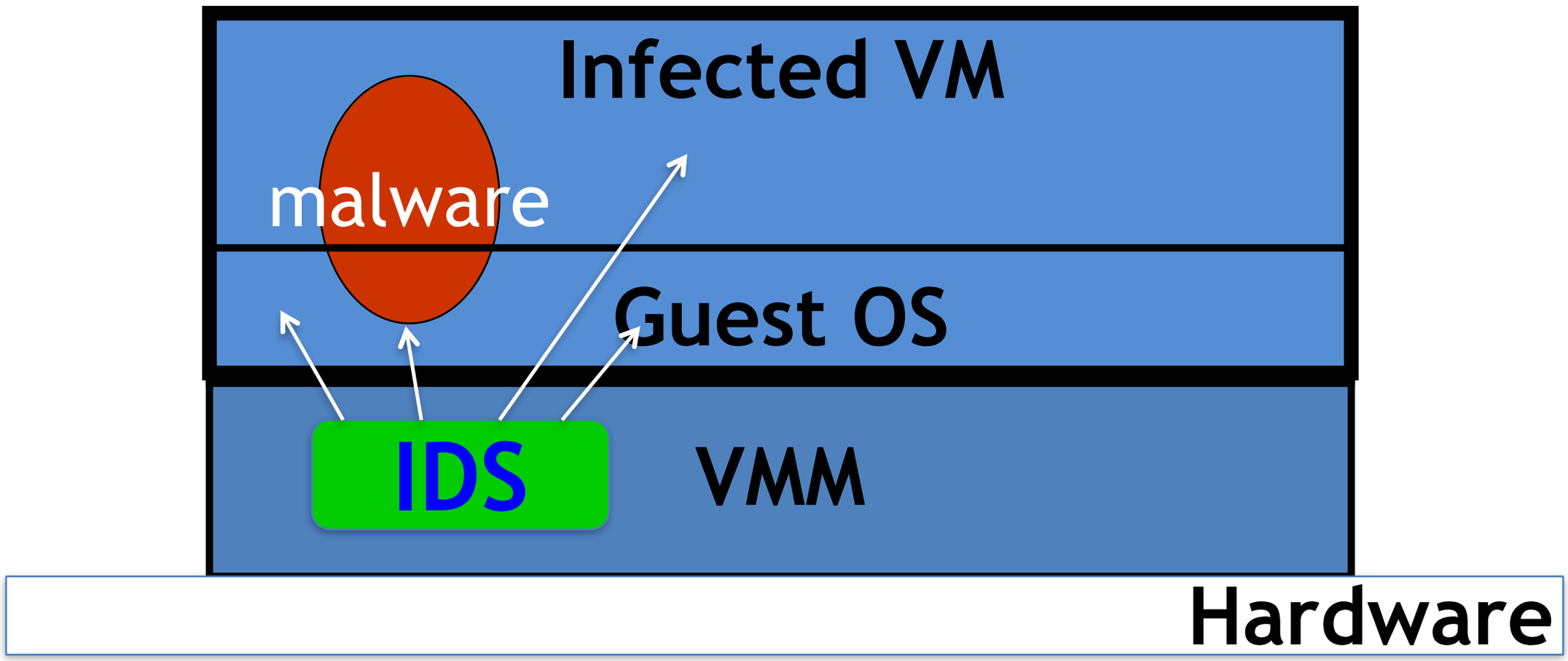


Example: Virtual Machines

VMM Introspection

Intrusion Detection / Anti-virus

- Runs as part of OS kernel and user space process
  - Kernel root kit can shutdown protection system
  - Common practice for modern malware
- Standard solution: run IDS system in the network
  - Problem: insufficient visibility into user's machine
- Better: run IDS as part of VMM (protected from malware)
  - VMM can monitor virtual hardware for anomalies
  - VMI: Virtual Machine Introspection
    - Allows VMM to check Guest OS internals


Infected VM
malware
Guest OS
IDS
VMM
Hardware

Sample checks

- Stealth root-kit malware:
  - Creates processes that are invisible to “ps”
  - Opens sockets that are invisible to “netstat”
- 1. Lie detector check
  - Goal: detect stealth malware that hides processes and network activity
  - Method:
    - VMM lists processes running in GuestOS
    - VMM requests GuestOS to list processes (e.g. ps)
    - If mismatch: kill VM

Sample checks

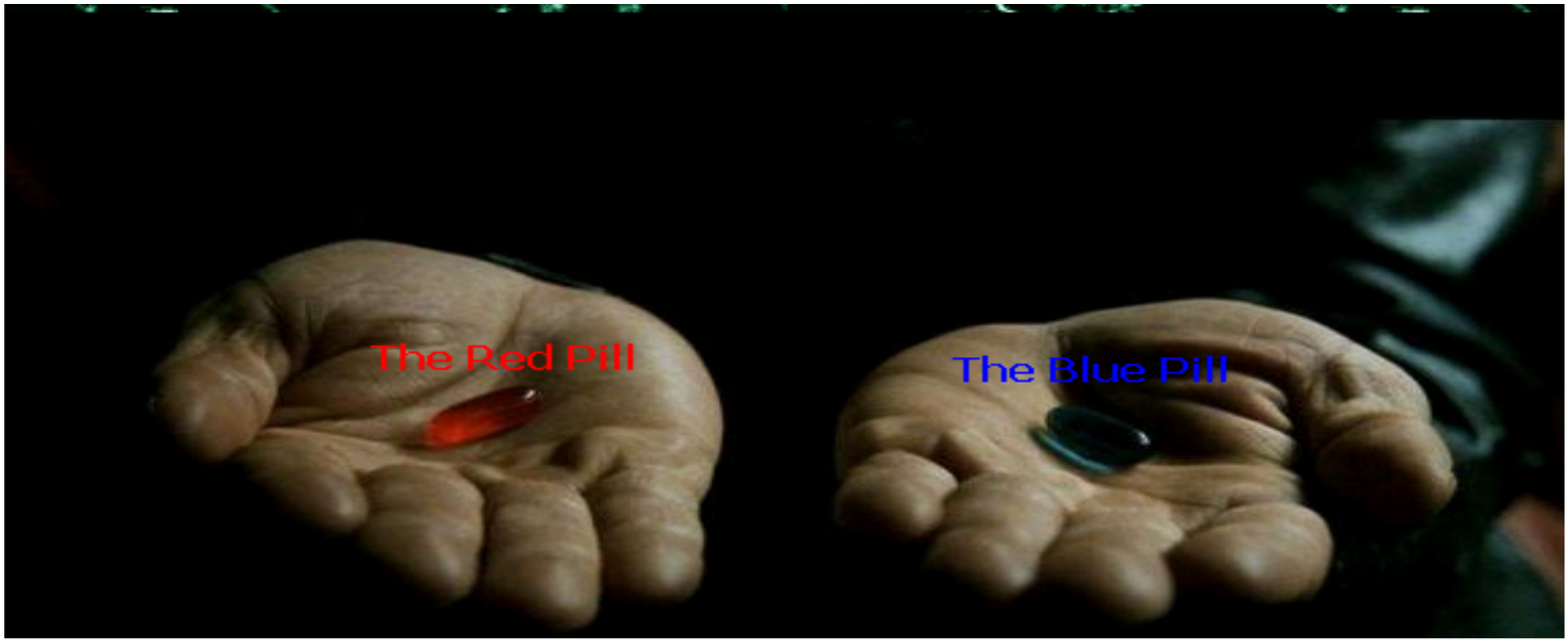
- 2. Application code integrity detector
  - VMM computes hash of user app code running in VM
  - Compare to whitelist of hashes
    - Kills VM if unknown program appears
- 3. Ensure GuestOS kernel integrity
  - example: detect changes to sys_call_table
- 4. Virus signature detector
  - Run virus signature detector on GuestOS memory

Example: Virtual Machines

Exploiting VM Isolation

The MATRIX

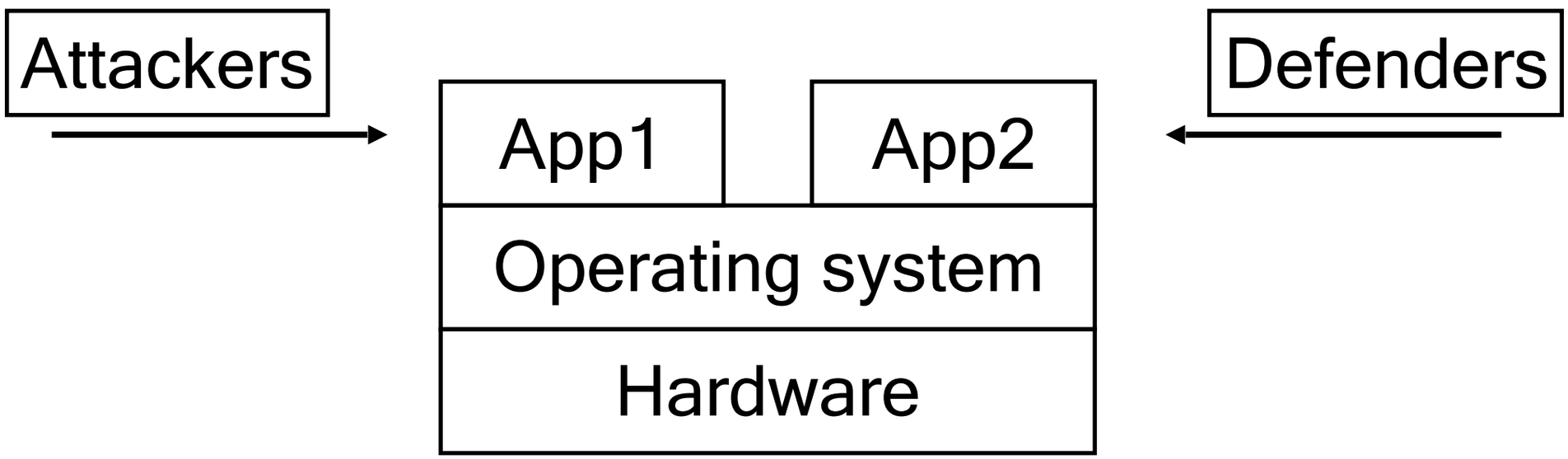

THE MATRIX IS ALL AROUND US IT IS THERE WHEN YOU WATCH TELEVISION
YOUR WALLPAPER.COM


The Red Pill
The Blue Pill

SubVirt: Implementing malware with virtual machines, King, Samuel T., and Peter M. Chen, IEEE Symposium on Security and Privacy (S&P'06), 2006

Motivation

- Attackers and defenders strive for control
  - Attackers monitor and perturb execution
    - Avoid defenders
  - Defenders detect and remove attacker
  - Control by lower layers

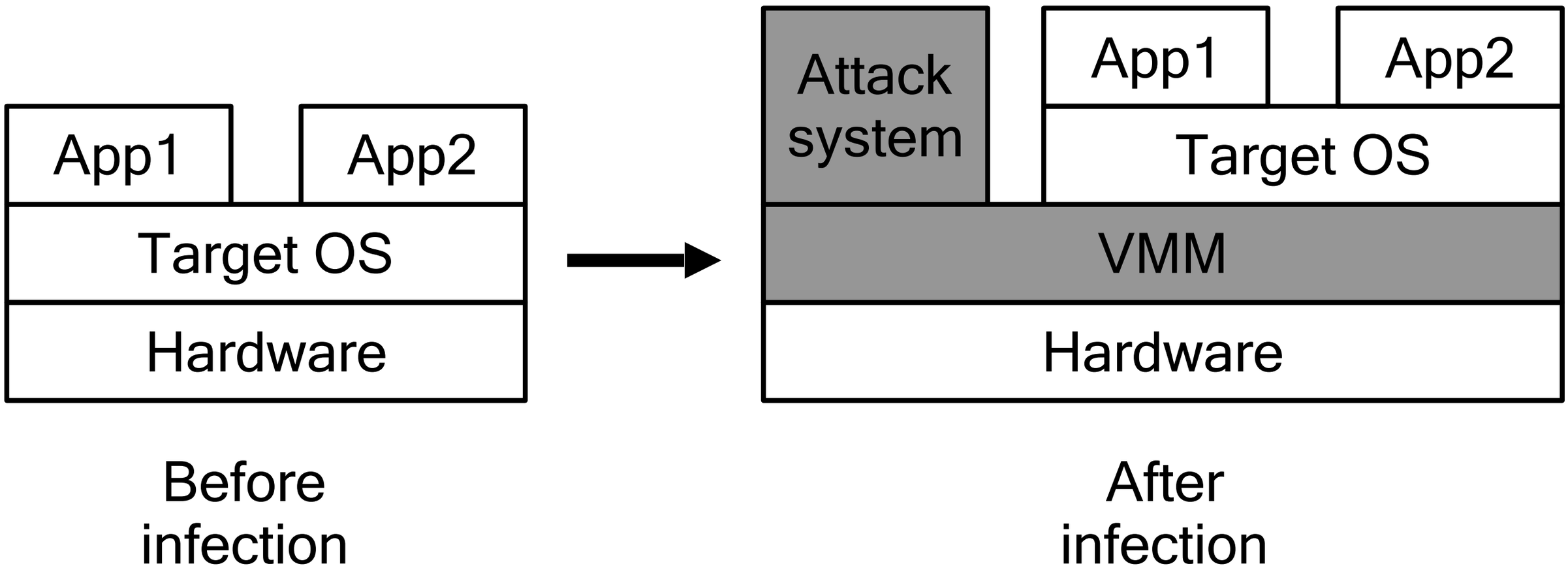


Virtual-machine based rootkits (VMBRs)

- VMM runs beneath the OS
  - Effectively new processor privilege level

- Fundamentally more control
- No visible states or events
- Easy to develop malicious services

Virtual-machine based rootkits (VMBRs)



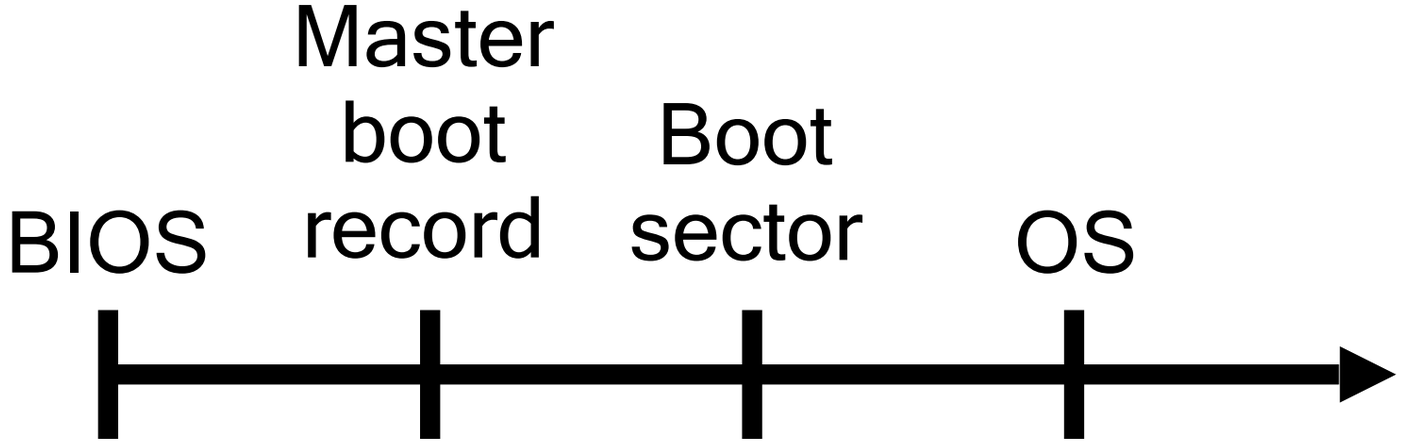
Installation

- Assume attacker has kernel privilege
  - Traditional remote exploit
  - Bribe employee
  - Malicious bootable CD-Rom

- Install during shutdown
  - Few processes running
  - Efforts to prevent notification of activity

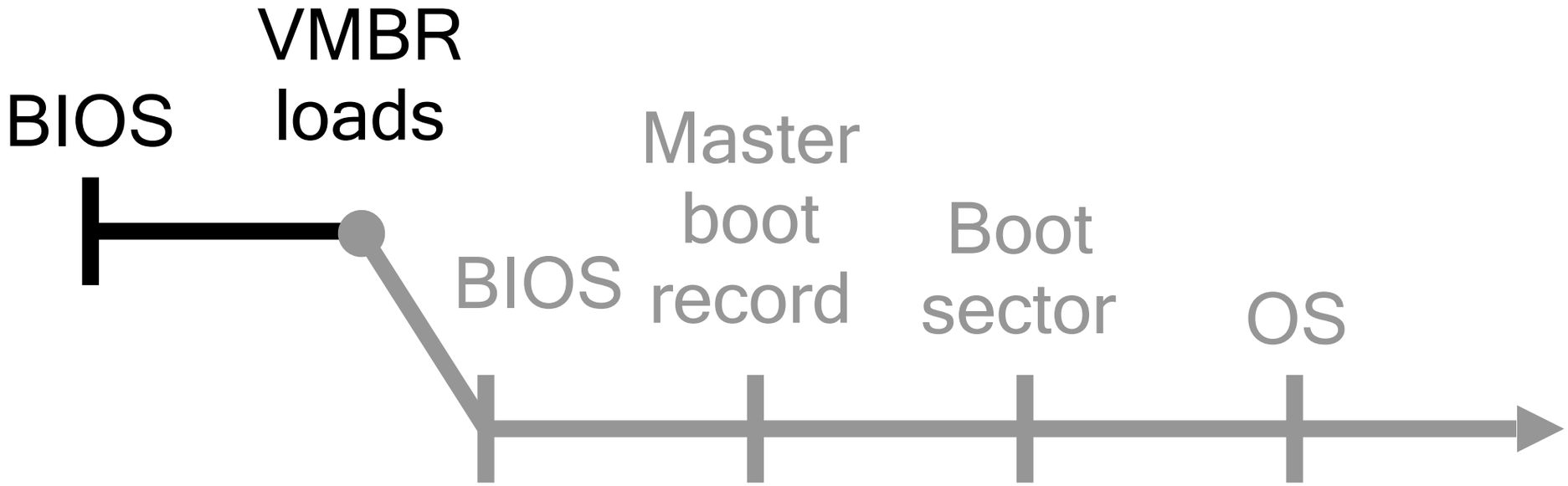
Installing a VMBR

- Modify the boot sequence



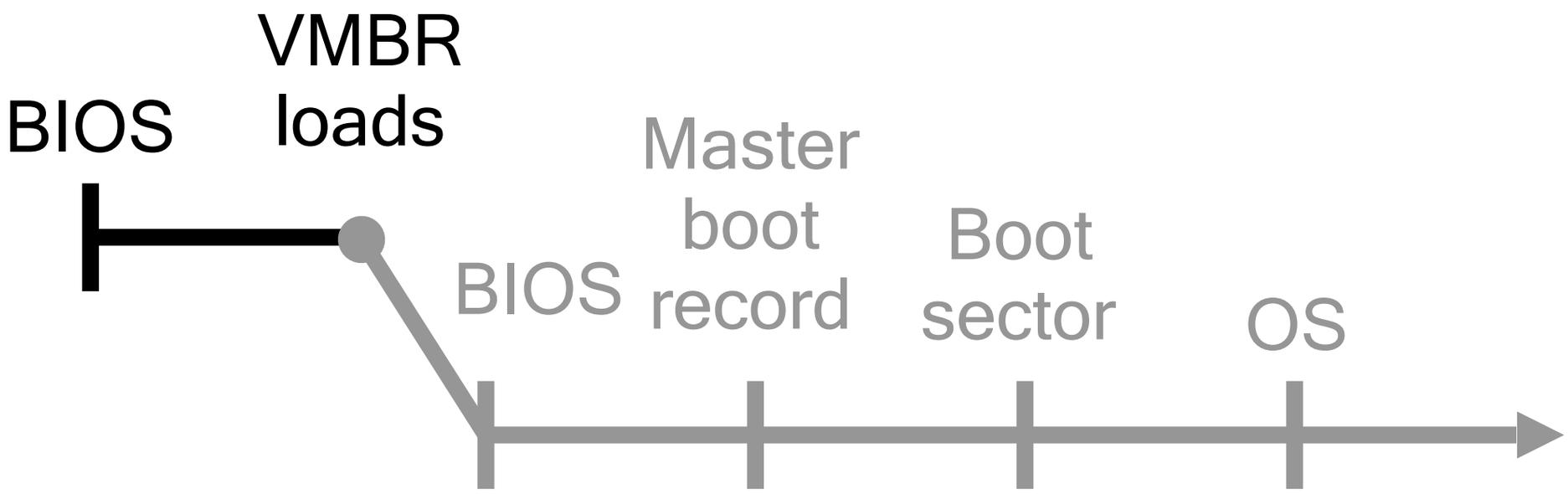
Installing a VMBR

- Modify the boot sequence



Maintaining control

- Hardware reset VMBR loses control
- Illusion of reset w/o losing control
- Reboot easy, shutdown harder

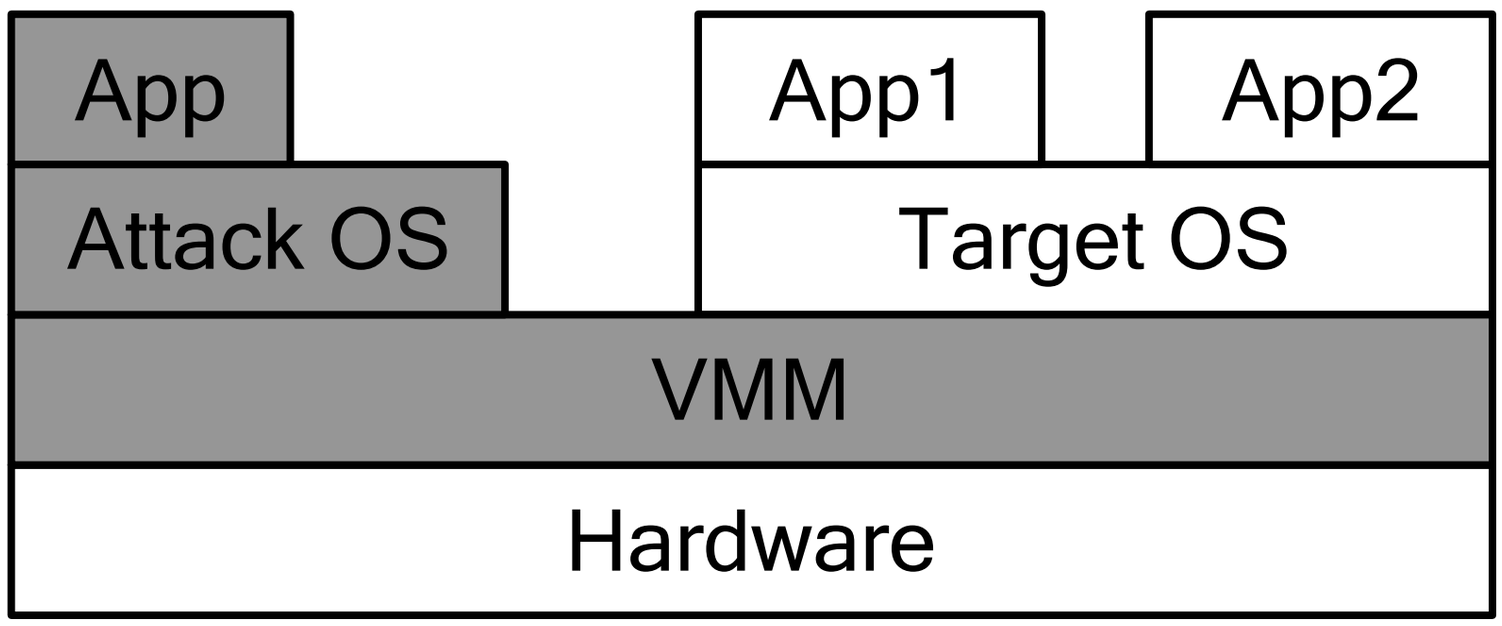


Maintaining control

- ACPI BIOS used for low power mode
  - Spin down disks
  - Display low power mode
  - Change power LED
- Illusion of power off, emulate shutdown
  - Control the power button
- System functionally unchanged

Malicious services

- Advantages of high and low layer malware
  - Provides low layer implementation
  - Still easy to implement services
- Use a separate attack OS to implement



Malicious services

- Zero interaction malicious services
  - E.g., phishing web server
- Passive monitoring
  - E.g., keystroke logger, file system scanner
- Active execution modifications
  - E.g., defeat VM detection technique

- All easy to implement

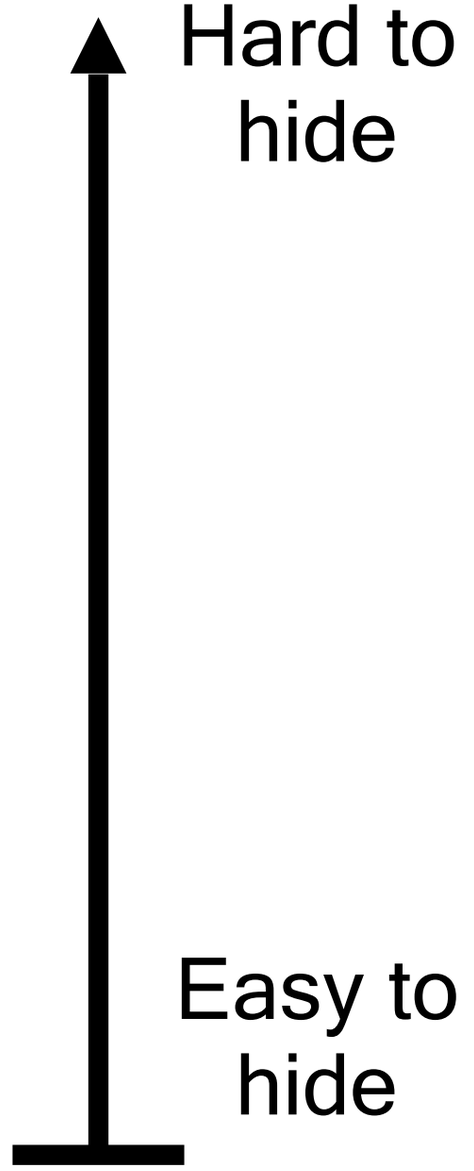
Defending against VMBRs

- Detecting VMBRs
  - Perturbations

- Where to run detection software

VMBR perturbations

- Inherent
  - Timing of key events
  - Space
- Hardware artifacts
  - Device differences
  - See paper for more details
- Software artifacts
  - VM icon
  - Device names



Security software above

- Attack state not visible
  - Can only detect side effects, e.g., timing

- VMBR can manipulate execution
  - Clock controlled by VMBR
  - Prevent security service from running
  - Turn off network
  - Disable notification of intrusion

Security software below

- More control, direct access to resources
  - Could detect states or events

- Secure VMM and/or secure hardware
- Boot from safe medium
  - Unplug machine from wall

Proof-of-concept VMBRs

- VMware / Linux host
- Virtual PC / Windows XP host
- Host OS was attack OS
- Malware payload ~100MB compressed

Proof-of-concept VMBRs

- Implemented four malicious services
  - Phishing web server
  - Keystroke logger + password parser
  - File system scanner
  - Countermeasure to detection tool
- Installation scripts and modules
- ACPI shutdown emulation
  - Both sleep states and power button control

Subverting VistaTM Kernel For Fun And Profit,

Joanna Rutkowska, Advanced Malware Labs, Black Hat 2006

Content

- Part I
  - Loading unsigned code into Vista Beta 2 kernel (x64) without reboot

- Part II
  - Blue Pill – creating undetectable malware on x64 using Pacifica technology

Part I – getting into the kernel

Signed Drivers in Vista x64

- All kernel mode drivers must be signed
- Vista allows to load only signed code into kernel
- Even administrator can not load unsigned module!
- This is to prevent kernel malware and anti-DRM
- Mechanism can be deactivated by:
  - attaching Kernel Debugger (reboot required)
  - Using F8 during boot (reboot required)
  - using BCDEdit (reboot required, will not be available in later Vista versions)
- This protection has been for the first time implemented in Vista Beta 2 build 5384.

How to bypass?

- Vista allows usermode app to get raw access to disk (provided they run with admin privileges of course)
  - `CreateFile(\\.\C:)`
  - `CreateFile(\\.\PHYSICALDRIVE0))`
- This allows us to read and write disk sectors which are occupied by the pagefile
- So, we can modify the contents of the pagefile, which may contain the code and data of the paged kernel drivers!
- No undocumented functionality required – all documented in SDK :)

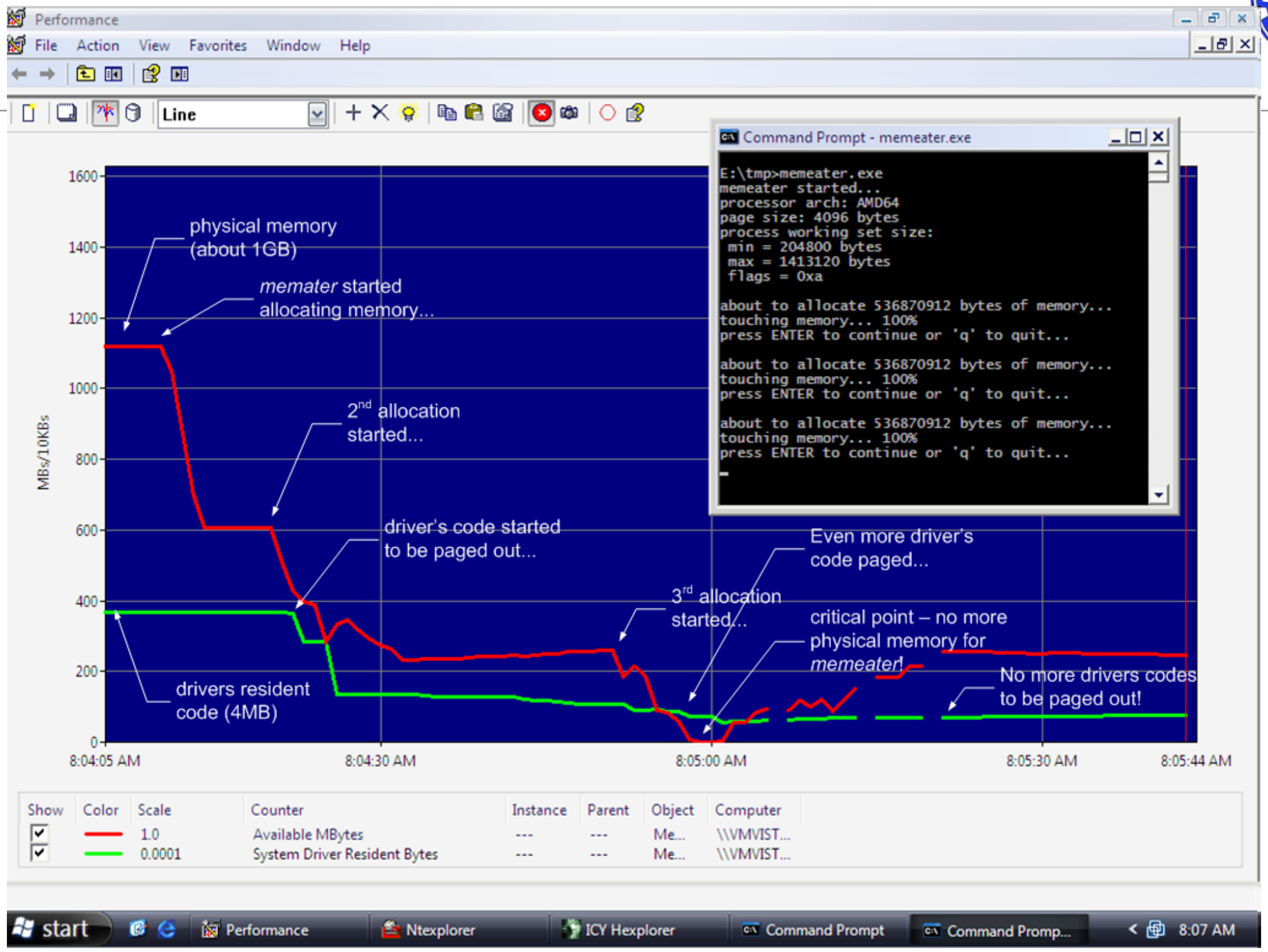
Challenges

- How to make sure that the specific kernel code is paged out to the pagefile?
- How to find that code inside pagefile?
- How to cause the code (now modified) to be loaded into kernel again?
- How to make sure this new code is executed by kernel?

How to force drivers to be paged?

- Allocate *lots of* memory for a process (e.g. using `VirtualAlloc()`)
- The system will try to do its best to back up this memory with the actual physical pages
- At some point there will be no more physical pages available, so the system will try to page out some unused code…
- Guess what is going to paged now… some unused drivers :)

Eating memory…



What could be paged?

- Pageable sections of kernel drivers (recognized by the section name starting with ‘PAGE’ string)
- Driver’s data allocated from a Paged pool (e.g. `ExAllocatePool()`)

Finding a target

- We need to find some rarely used driver, which has some of its code sections marked as pageable…
- How about NULL.SYS?
- After quick look at the code we see that its dispatch routine is located inside a PAGE section – one could not ask for more :)
- It should be noted that there are more drivers which could be used instead of NULL – finding them all is left as an exercise to the audience ;)

Locating paged code inside pagefile

- This is easy – we just do a pattern search
  - if we take a sufficiently long binary string (a few tens of bytes) its very unlikely that it will appear more then once in a page file
- Once we find a pattern we just replace the first bytes of the dispatch function with our shellcode

How to make sure our shellcode gets executed?

- We need to ask kernel to be kind enough and execute our driver's routine (whose code we have just replaced in pagefile)
- In case of replacing driver's dispatch routine it's just enough to call `CreateFile()` specifying the target driver's object to be opened
- This will cause the driver's paged section to be loaded into memory and then executed!

Putting it all together

- Allocate lots of memory to cause unused drivers code to be paged
- Replace the paged out code (inside pagefile) with some shellcode
- Ask kernel to call the driver code which was just replaced

Part II – Blue Pill

Invisibility by Obscurity

- Current malware is based on a concept...
- e.g. FU unlinks EPROCESS from the list of active processes in the system
- e.g. deepdoor modifies some function pointers inside NDIS data structures
- … etc…
- Once you know the concept you can write a detector!
- This is boring!

Imagine a malware…

- …which does not rely on a concept to remain undetected…
- …which can not be detected, even though its algorithm (concept) is publicly known!
- …which can not be detected, even though it’s code is publicly known!

Blue Pill Idea

- Exploit AMD64 SVM extensions to move the operating system into the virtual machine (do it ‘on-the-fly’)
- Provide thin hypervisor to control the OS
- Hypervisor is responsible for controlling “interesting” events inside gust OS

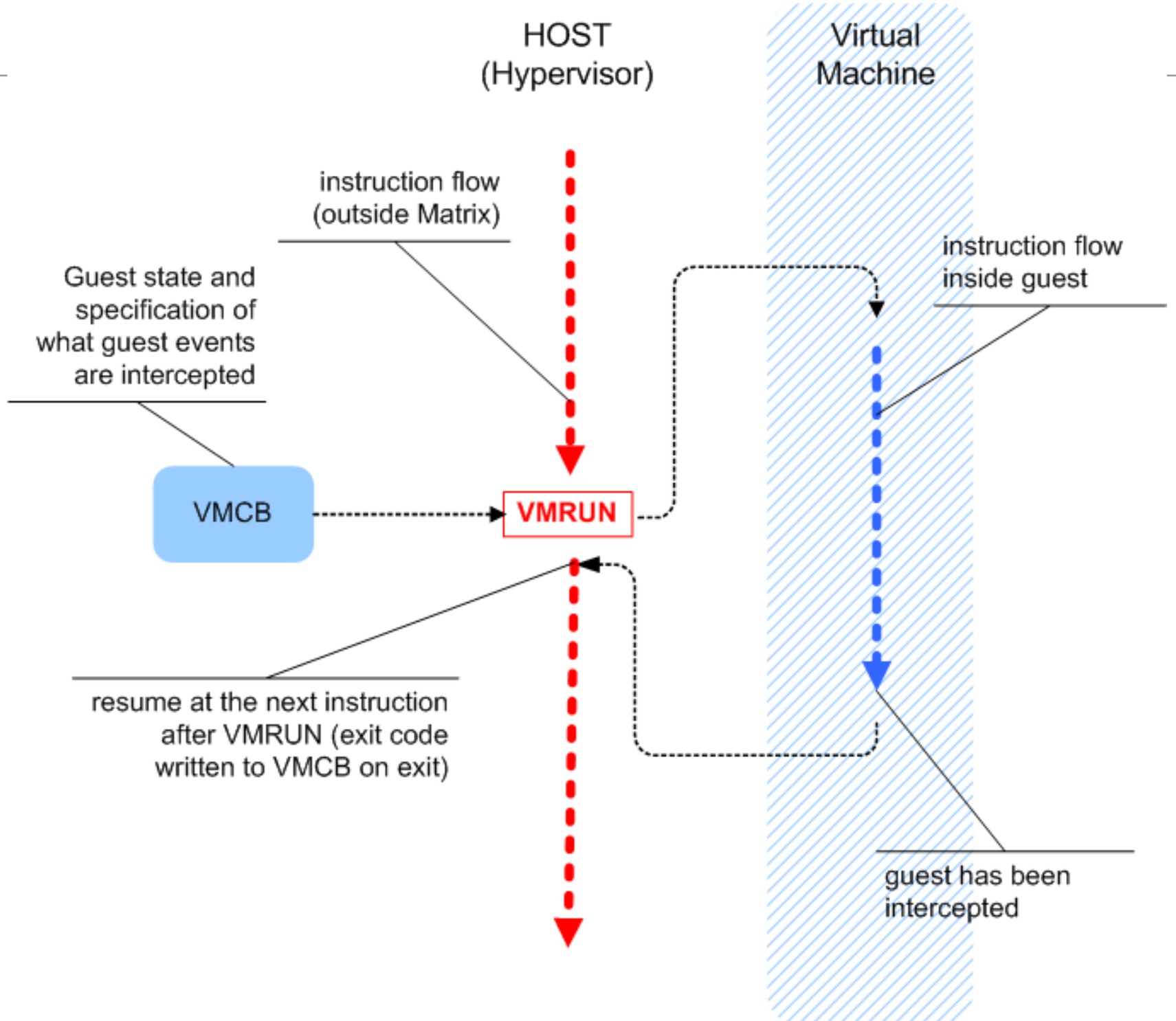
AMD64 & SVM

- Secure Virtual Machine (AMD SVM) Extensions (AKA Pacifica)
- May 23rd, 2006 – AMD releases Athlon 64 processors based on socket AM2 (revision F)
- AM2 based processors are the first to support SVM extensions
- AM2 based hardware is available in shops for end users as of June 2006

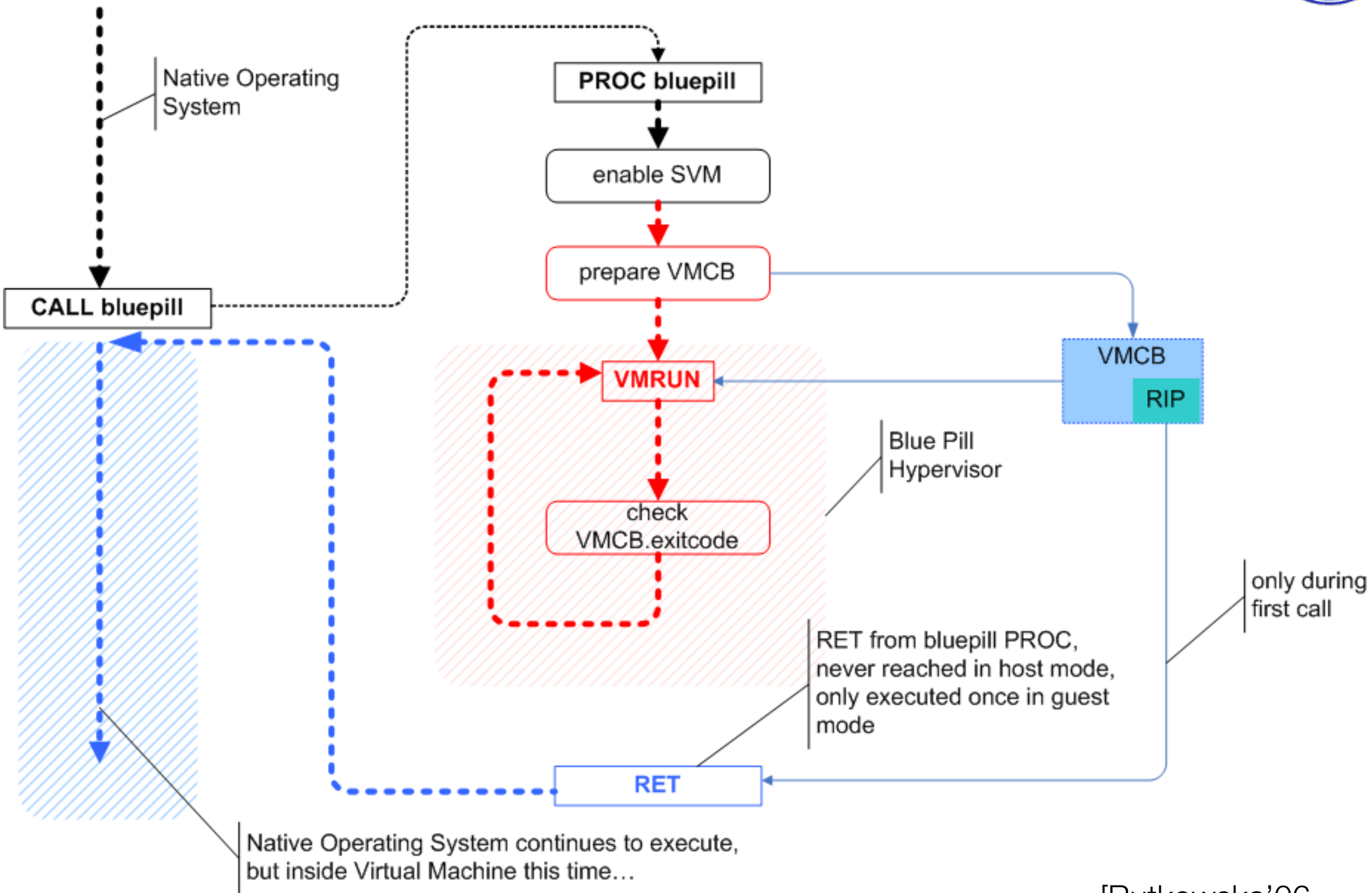
SVM

- SVM is a set of instructions which can be used to implement Secure Virtual Machines on AMD64
- MSR EFER register: bit 12 (SVME) controls weather SVM mode is enabled or not
- EFER.SVME must be set to 1 before execution of any SVM instruction.
- Reference:
  - AMD64 Architecture Programmer's Manual Vol. 2: System Programming Rev 3.11
  - http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

The heart of SVM: VMRUN instruction



Blue Pill Idea (simplified)



[Rutkowska'06, COSEINC Research]

BP installs itself ON THE FLY!

- The main idea behind BP is that it installs itself on the fly
- Thus, no modifications to BIOS, boot sector or system files are necessary
- BP, by default, does not survive system reboot
- But this is not a problem:
  - servers are rarely restarted
    - In Vista the ‘Power Off’ button does not shut down the system – it only puts it into stand by mode!
- And also we can intercept (this has not been yet implemented):
  - restart events (hypervisor survives the reboot)
  - shutdown events (emulated shutdown)

SubVirt Rootkit

- SubVirt has been created a few months ago by researches at MS Research and University of Michigan
- SubVirt uses commercial VMM (Virtual PC or VMWare) to run the original OS inside a VM

SubVirt vs. Blue Pill

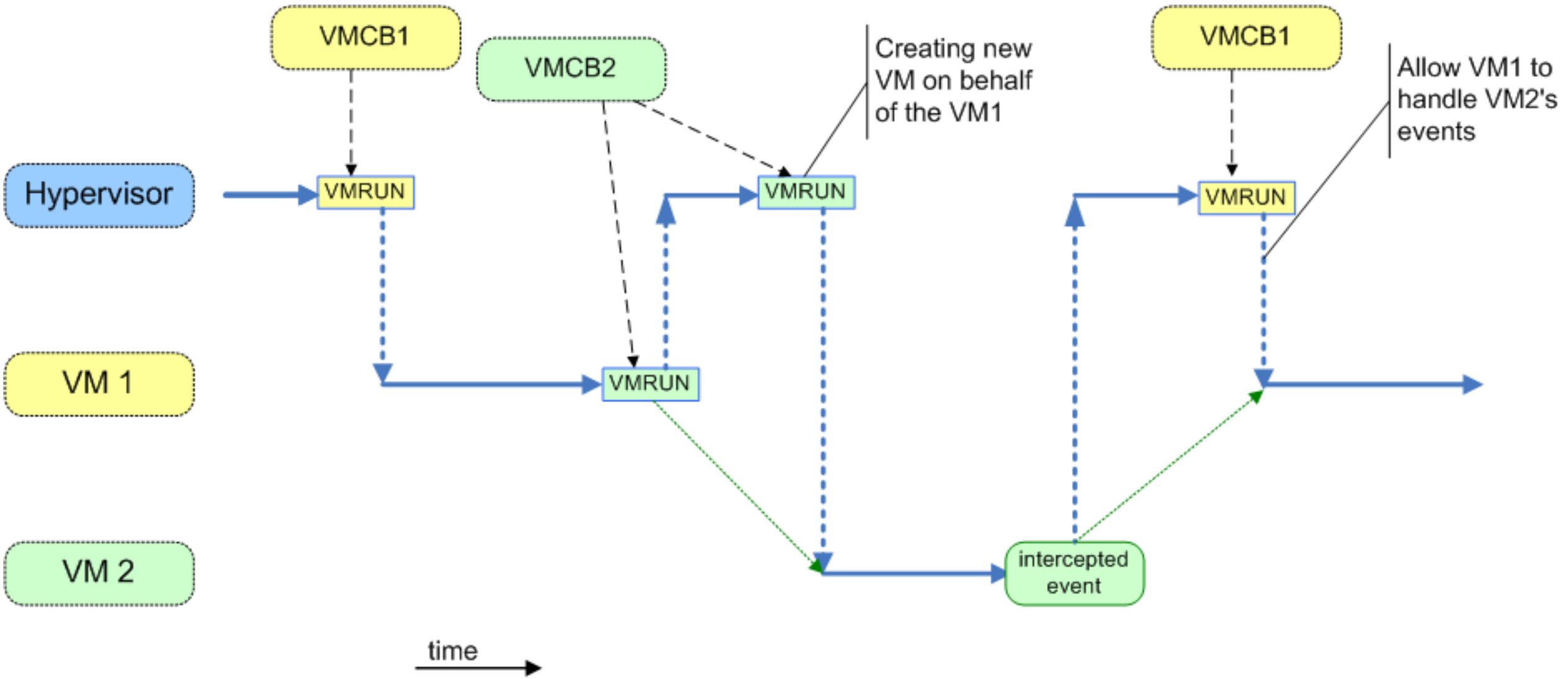
- SV is permanent! SV has to take control before the original OS during the boot phase. SV can be detected off line.
- SV runs on x86, which does not allow for full virtualization
- SV is based on a commercial VMM, which creates and emulates virtual hardware. This allows for easy detection

- Blue Pill can be installed on the fly – no reboot nor any modifications in BIOS or boot sectors are necessary. BP can not be detected off line.
- BP relies on AMD SVM technology which promises full virtualization
- BP uses ultra thin hypervisor and all the hardware is natively accessible without performance penalty

Matrix inside another Matrix

- What happens when you install Blue Pill inside a system which is already bluepilled?
- If nested virtualization is not handled correctly this will allow for trivial detection – all the detector would have to do was to try creating a test VM using a VMRUN instruction
- Of course we can cheat the guest OS that the processor does not support SVM (because we control MSR registers from hypervisor), but this wouldn't cheat more inquisitive users ;)
- So, we need to handle nested VMs…

Nested VMs


VMCB1
VMCB2
Creating new VM on behalf of the VM1
VMCB1
Allow VM1 to handle VM2's events
Hypervisor
VMRUN
VMRUN
VMRUN
VM 1
VMRUN
VM 2
intercepted event
time

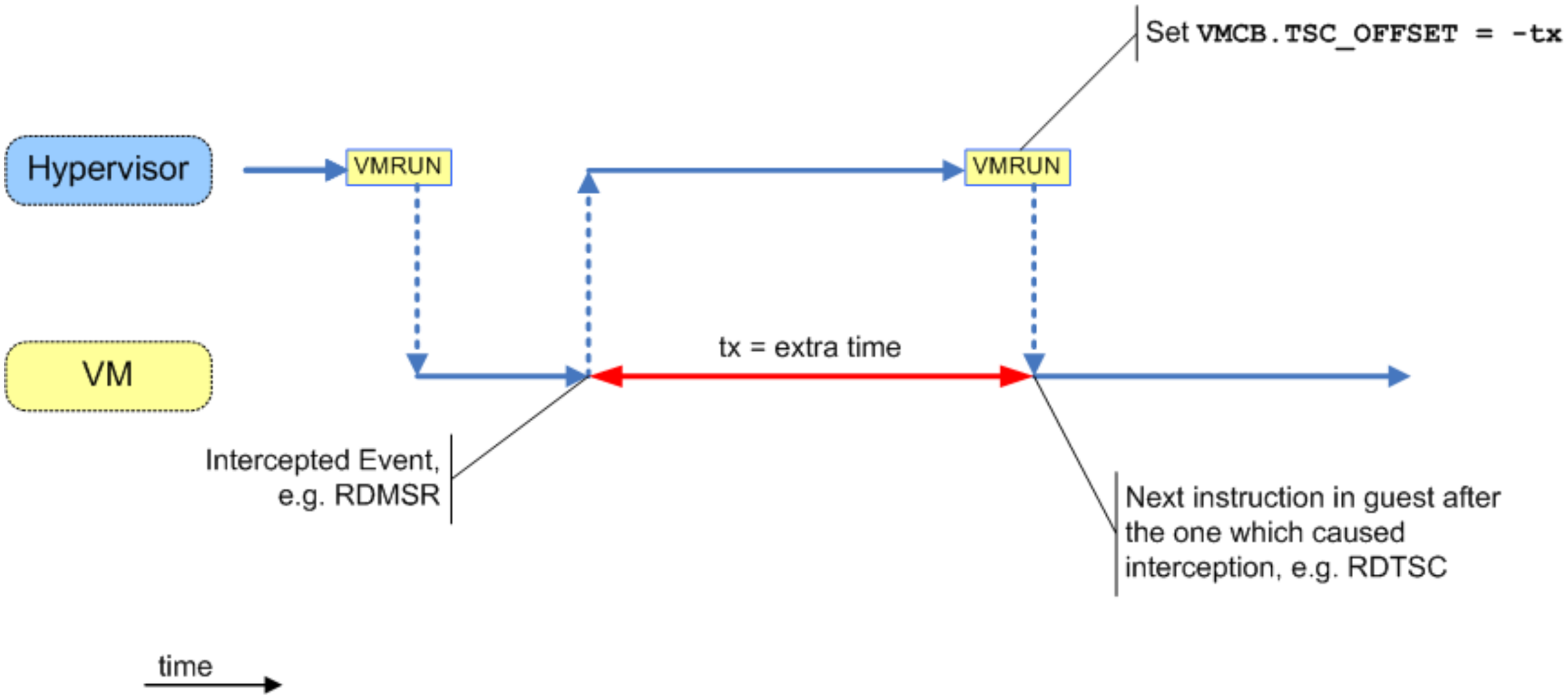
Detection via timing analysis

- We can assume that some of the instructions are always intercepted by the hypervisor
  - `VMMCALL`
  - `RDMSR` – to cheat about the value of `EFER.SVME` bit
- So, not surprisingly, the time needed to execute `RDMSR` to read the value of EFER would be different (longer) when running from guest
- Detector can execute such instructions a few millions of times and measure the time.

Cheating timing analysis

- The first problem is that detector (usually) does not have a base line to compare the results with…
- But even if it had (e.g. when testing access to physical pages and while one of them being backed up by disk) still we can cheat it!
- SVM offers a way to fool the guest's time stamp counter (obtained using RDTSC).
- All we have to do is to adjust VMCB.TSC_OFFSET accordingly before executing VMRUN (which resumes the guest)

Time dilatation for guest



Getting the real time…

Time profiling in practice

- Now imagine that you need to check 1000 computers in your company using the “external” stopwatch…
- Now imagine that you need to do this a couple of time every day…
- Time dilatation should make it impossible to write a self sufficient detector based on timing analysis!

- The challenge: we need a good ‘calibrating’ mechanism so that we know how much time to subtract.

Pacifica vs. Vanderpool

- Pacifica (SVM) and Vanderpool (VT-x) are not binary compatible
- However they seem to be very similar
- XEN even implements a common abstraction layer for both technologies
- It seems possible to port BP to Intel VT-x

Android Security Architecture

Introduction

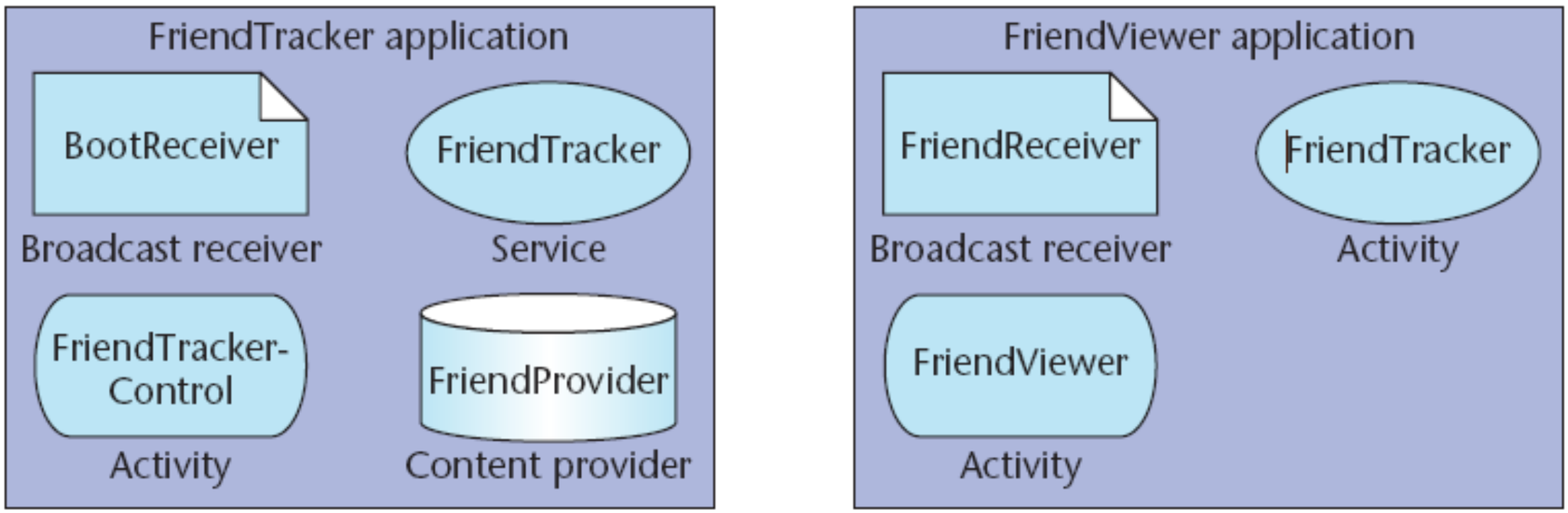
- Next generation open operation system will be developed on small mobile devices.

- Android (Google)
  - a widely anticipated open source operating system for mobile devices
  - it provide base operation system, application middleware layer, Java software development kit and a collection of system applications.

Introduction (cont.)

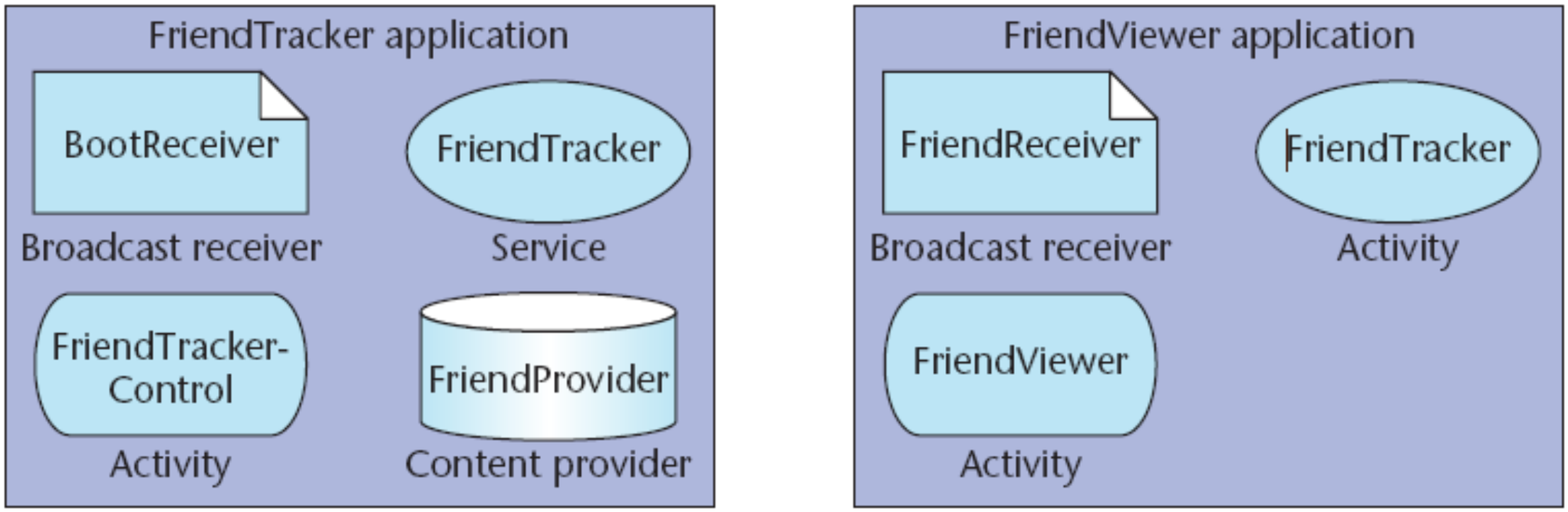
- Feature of Android
  - Doesn’t support applications developed for other platforms
  - Restricts application interaction to its special APIs by running each application as its own user identity
  - Uses a simple permission label assignment model to restrict access to resources and other applications

Android Applications --- Example



- Example of location-sensitive social networking application for mobile phones in which users can discover their friends locations.
- **Activities** provide a user interface, **Services** execute background processing, **Content providers** are data storage facilities, and **Broadcast receivers** act as mailboxes for messages from other applications.

Android Applications --- Example Application(cont.)

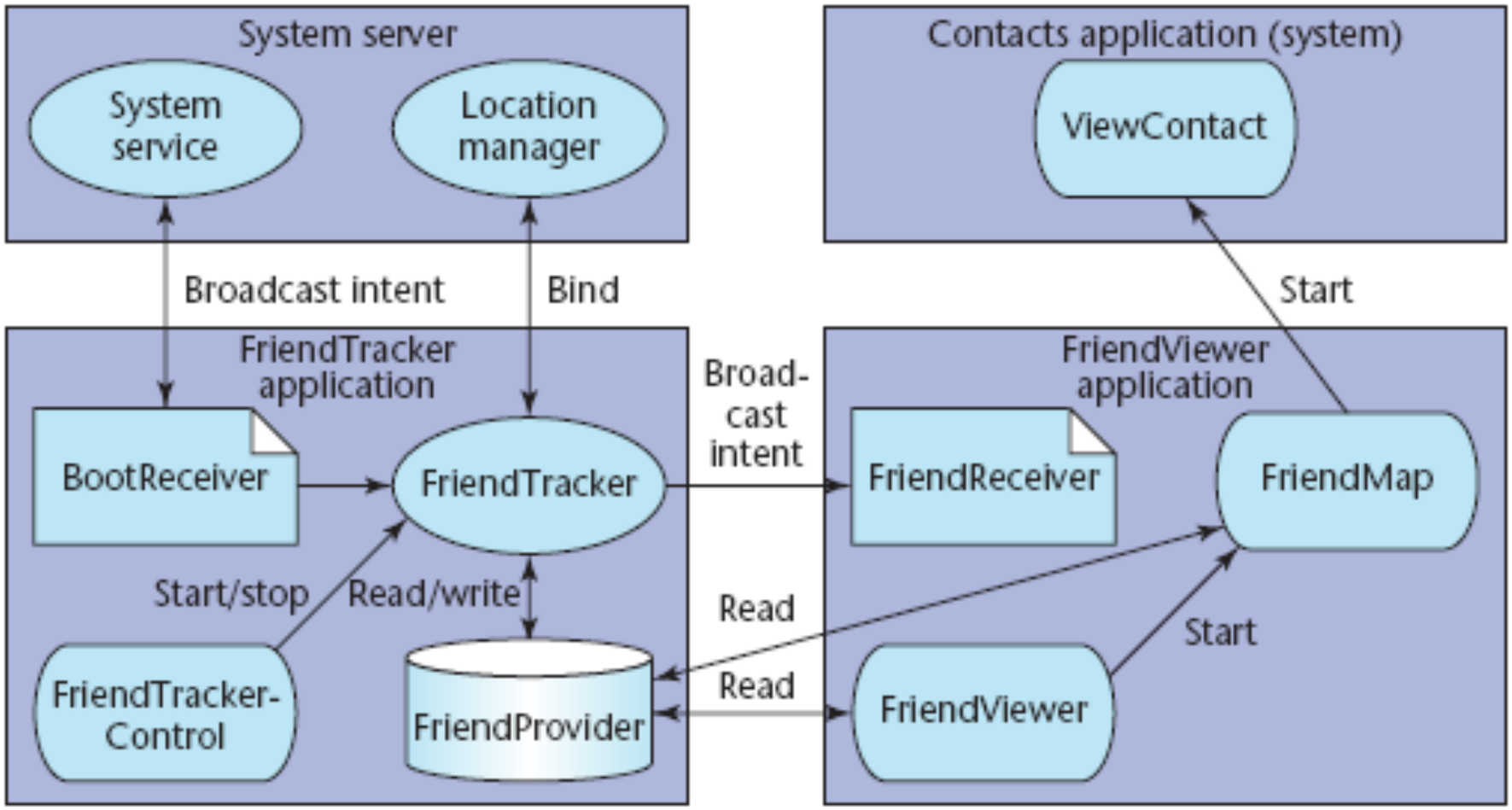


- Take FriendTracker application for example,
- FriendTracker (**Service**) polls an external service to discover friends locations
- FriendProvider (**Content provider**) maintains the most recent geographic coordinates for friends
- FriendTrackerControl (**Activity**) defines a user interface for starting and stopping the tracking functionality
- BootReceiver (**Broadcast receiver**) gets a notification from the system once it boots (app uses this to automatically start the FriendTracker service).

Android Applications--- Component Interaction

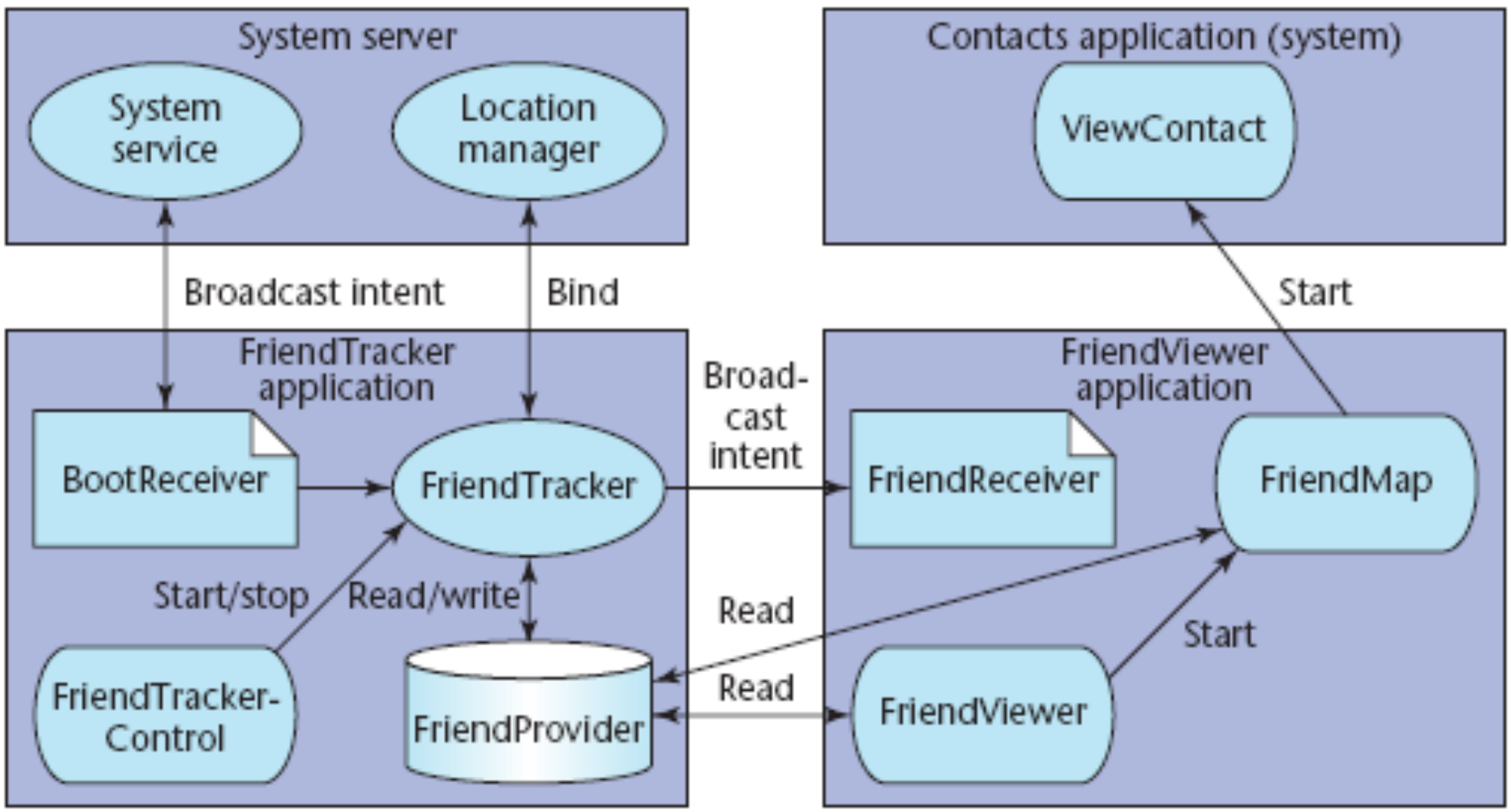
- Intent - is the primary mechanism for component interaction, which is simply a message object containing a destination component address and data
- Action - the process of inter-components communication
- The Android API defines methods that accept intents and uses that information to
  - start activities (startActivity(Intent))
  - start services (startService (Intent))
  - broadcast messages (sendBroadcast(Intent)).
  - The invocation of these methods tells the Android framework to begin executing code in the target application.
- In other words: an intent object defines the “intent” to perform an “action.”

Android Applications--- Component Interaction (cont.)



- Example: Interaction between components in applications and with components in system applications. Interactions occur primarily at the component level.

Android Applications--- Component Interaction (cont.)

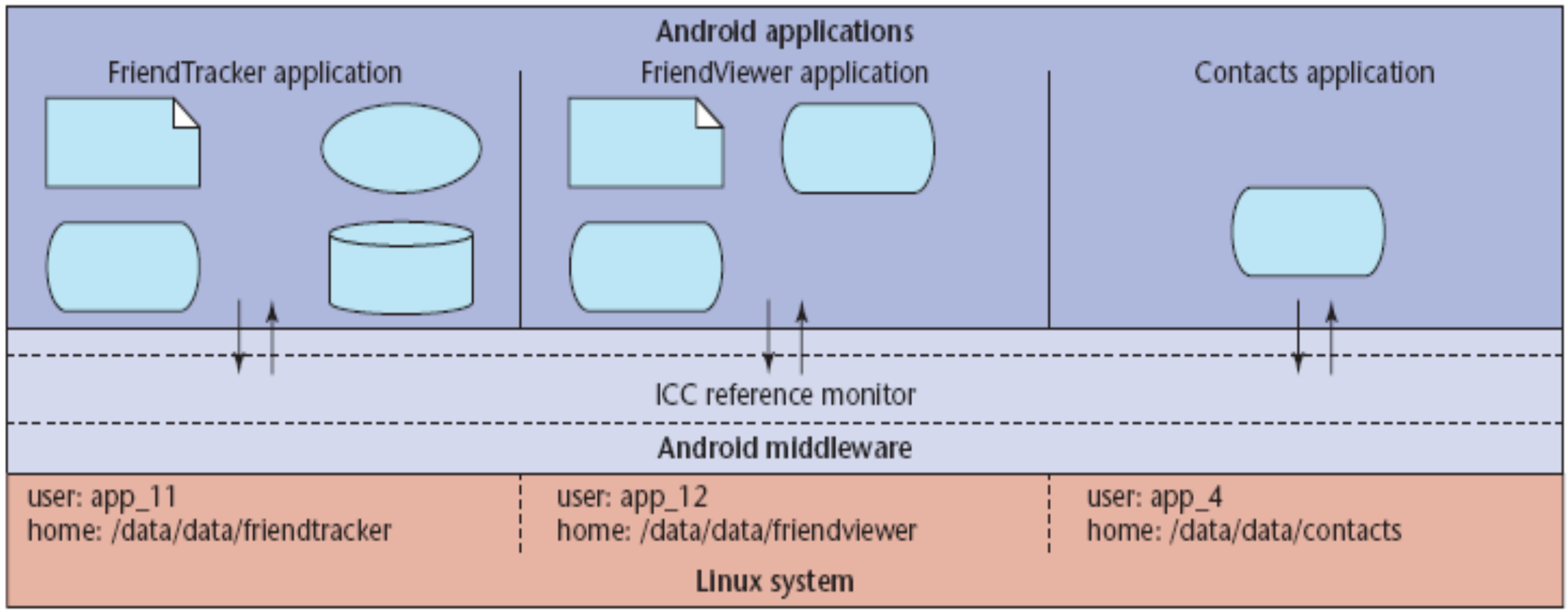


- Each component type supports interaction specific to its type. For example, Service components support start , stop, and bind actions, so the FriendTrackerControl (Activity) can start and stop the FriendTracker (Service) that runs in the background.

Security Enforcement

- Android protect application at system level and at the Inter-component communication (ICC) level. This article focus on the ICC level enforcement.
- Each application runs as a unique user identity, which lets Android limit the potential damage of programming flaws.

Security Enforcement (cont.)

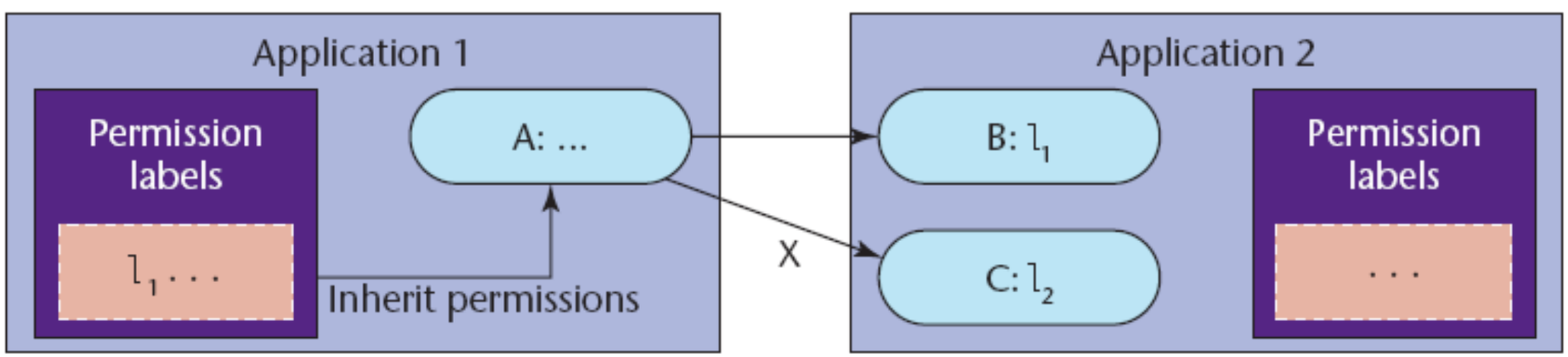


Example: Protection. Security enforcement in Android occurs in two places: each application executes as its own user identity, allowing the underlying Linux system to provide system-level isolation; and the Android middleware contains a reference monitor that mediates the establishment of inter-component communication (ICC).

Security Enforcement (cont.)

- Core idea of Android security enforcement - labels assignment to applications and components
- A reference monitor provides mandatory access control (MAC) enforcement of how applications access components.
- Access to each component is restricted by assigning it an access permission label; applications are assigned collections of permission labels.
- When a component initiates ICC, the reference monitor looks at the permission labels assigned to its containing application and— if the target component's access permission label is in that collection— allows ICC establishment to proceed.

Security Enforcement (cont.)



- Example: Access permission logic. The Android middleware implements a reference monitor providing mandatory access control (MAC) enforcement about how applications access components. The basic enforcement model is the same for all component types. Component A's ability to access components B and C is determined by comparing the access permission labels on B and C to the collection of labels assigned to application 1.

Security Enforcement (cont.)

- Assigning permission labels to an application specifies its protection domain. Assigning permissions to the components in an application specifies an access policy to protect its resources.
- Android's policy enforcement is mandatory, all permission labels are set at install time and can't change until the application is reinstalled.
- Android's permission label model only restricts access to components and doesn't currently provide information flow guarantees.

Security Refinements --- Public vs. Private Components

- Applications often contain components that another application should never access.
  - For example, component related to password storing. The solution is to define private component.
- This significantly reduces the attack surface for many applications.

Security Refinements --- Implicitly Open Components

- At development time, if the decision of access permission is unclear, The developer can permit the functionality by not assigning an access permission to it.
- If a public component doesn’t explicitly have an access permission listed in its manifest definition, Android permits any application to access it.

Security Refinements --- Broadcast Intent Permissions

- Sending the unprotected intent is a privacy risk.
  - Other apps maybe listening in.
- Android API for broadcasting intents optionally allows the developer to specify a permission label to restrict access to the intent object.

Security Refinements --- Content Provider Permissions

- If the developer want his application to be the only one to update the contents but for other applications to be able to read them.
- Android allows such a security policy assigning read or write permissions.

Security Refinements --- Protected APIs

- Not all system resources(for example, network) are accessed through components—instead, Android provides direct API access.
- Android protects these sensitive APIs with additional permission label checks:
  - an application must declare a corresponding permission label in its manifest file to use them.

Security Refinements --- Permission Protection Levels

- The permission protection levels provide a means of controlling how developers assign permission labels:
  - **Normal**
    - for legacy support, equivalent to old application permission type
  - **Dangerous** permission granted after user confirmation
  - **Signature** permissions ensure that only the framework developer can use the specific functionality
    - only Google applications can directly interface the telephony API, for example
  - **Signature or System**
    - for legacy support, equivalent to old system permission type

Security Refinements --- Pending Intents

- Pending intent:
  - A developer defines an intent object to perform an action.
  - How ever, instead of performing the action, the developer passes the intent to a special method that creates a PendingIntent object corresponding to the desired action.
    - The PendingIntent object is simply a reference pointer that can pass to another application.
    - when intent is invoked, it causes a RPC with the original application, in which the ICC executes with all its permissions
- Pending intents allow applications included with the framework to integrate better with third-party applications.

Lessons in Defining Policy

- Android security policy begins with a relatively easy-to-understand MAC enforcement model, but the number and subtlety of refinements make it difficult to discover an application's policy.
- For example, how do you control access to permission labels?
  - Android's permission protection levels provide some control, but more expressive constraints aren't possible.
  - e.g. should an application be able to access both the microphone and the Internet?

Acknowledgments/References

- [Seshan'16] 15-440: Distributed Systems Syllabus, Yuvraj Agarwal, Srini Seshan, CMU, Fall 2016
- [Boneh'15] CS 155, Computer Security, Dan Boneh, Stanford University, 2015
- [King'06] SubVirt: Implementing malware with virtual machines, King, Samuel T., and Peter M. Chen, IEEE Symposium on Security and Privacy (S&P'06), 2006
- [Rutkowska'06, COSEINC Research] Subverting Vista Kernel For Fun And Profit, Joanna Rutkowska, Black Hat Briefing 2006. Copyright COSEINC Research, Advanced Malware Labs.
- [Wu'09, Enck'09] CS 585: Computer Security, Feng Zhu, University of Alabama in Huntsville, Spring 2009 AND Understanding Android Security, William Enck, Machigar Ongtang and Patrick Mcdaniel, IEEE Security and Privacy magazine 2009