CIS 4360
Secure Computer Systems

Virtual Machine Introspection for Intrusion Detection

Professor Qiang Zeng
Spring 2017

TEMPLE UNIVERSITY

Some slides are courtesy of Garfionkel and Rosenblum

Previous Class

- Sandboxing through separate processes
  - The crash of the NaCl process will not crash your Chrome tab process
- Sandboxing through static validator
  - The control can only jump to the set of instructions that have been well analyzed
  - You can never jump to the middle of an instruction
- Sandboxing through Software Fault Isolation
  - Classic SFI is a little bit slow
  - H/w assisted SFI causes ~0 overhead

Introduction(1/3)

- Two ways to defeat **Intrusion Detection System(IDS)**
  - Evasion
    - Disguising malicious activity
    - IDS failed to recognize it
  - Attack
    - Tampering with the IDS or components it trust

Introduction(2/3)

- Host-based Intrusion Detection System(HIDS)
  - Is integrated into the host it is monitoring as an application or a part of the OS
    - High visibility
  - IDS Crash
    - Cannot suspend the OS
      - Rely on OS to resume its operation
- Network-based Intrusion Detection System(NIDS)
  - Isolation from the host
    - High attack resistance
    - OS has been compromised-> remain visibility
  - IDS Crash
    - Suspend connectively

Introduction(3/3)

- **Virtual Machine Introspection(VMI)**
  - High visibility and high attack resistance
  - Livewire
  - Crash
    - Suspend monitored guest OS trivially
- Leveraging virtual machine monitor(VMM) technology
  - Pull VMI outside of the host
  - Directly inspect the hardware state of the virtual machine that a monitored host is running on
  - Interpose at the architecture interface of the monitored host

VMM and VMI(1/3)

- VMM = Hypervisor
  - VMM is a thin layer of software that runs directly on the hardware of a machine
  - Export a virtual machine abstraction that resembles the underlying hardware
- Guest OS
  - The OS running inside of a VM
- Guest Application
  - Applications running on guest OS

VMM and VMI(2/3)

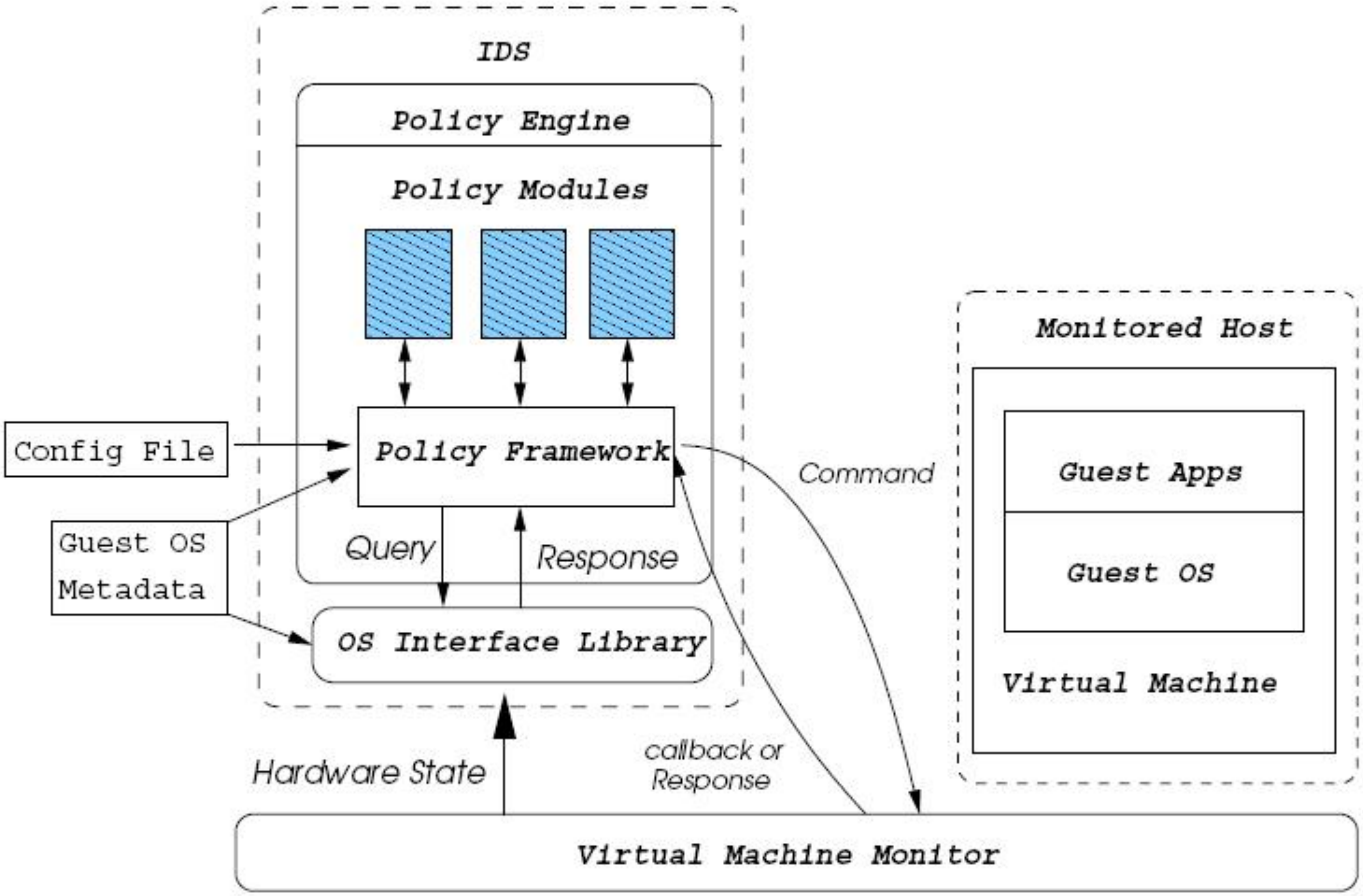
- VMM is difficult for an attacker to compromise
  - Simple-enough that we can reasonably hope to implement it correctly
    - The interface for VMM is significant simpler than OS
    - The protection model is significant simpler than OS
      - No concerns about control sharing
    - 30K lines of code
      - Lack of file system, network stack, a full fledged virtual memory system

VMM and VMI(3/3)

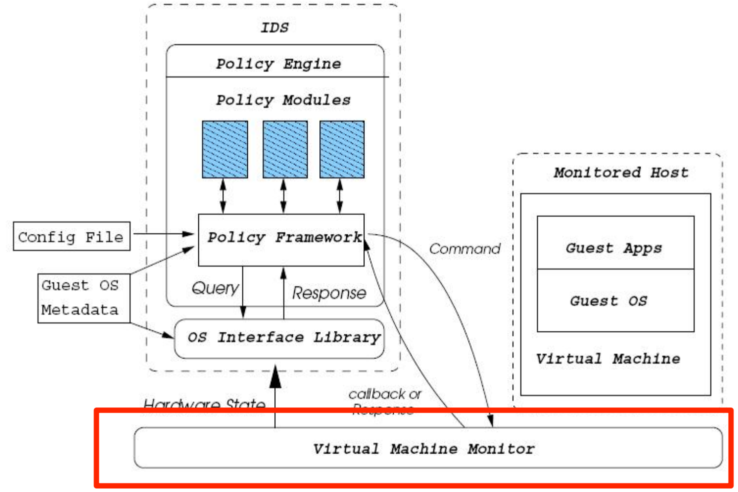
- VMI IDS leverages three properties of VMMs
  - **Isolation**
    - Software running in a VM cannot access or modify the software running in the VMM or in a separate VM
    - If a VM was completely subverted - > intruder cannot tamper with the IDS
  - **Inspection**
    - VMM has access to all the state of a VM
      - CPU state, all memory, all I/O device state
    - Difficult to evade a VMI IDS since there is no state in the monitored system that the IDS cannot see
  - **Interposition**
    - VMM can interpose on certain VM operations(e.g executing privileged instructions)

Design

-

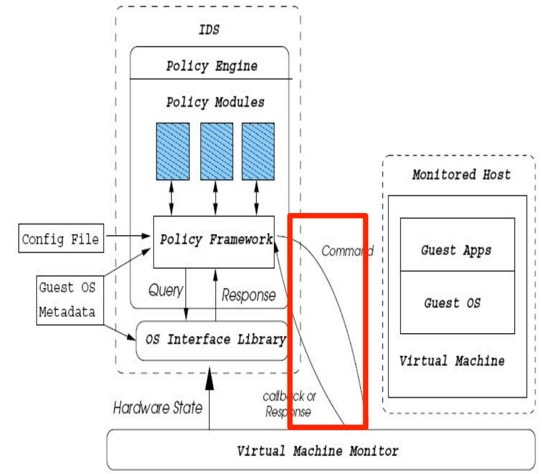

IDS
Policy Engine
Policy Modules
Config File
Policy Framework
Guest OS
Metadata
Query
Response
OS Interface Library
Command
Monitored Host
Guest Apps
Guest OS
Virtual Machine
Hardware State
callback or
Response
Virtual Machine Monitor

Design- VMM



- Provide isolation by default
- Inspection and Interposition
  - Require some modification of the VMM
- Trade off
  - Functionality vs. Simplicity
    - Can provide significant benefits but IDS will be exposed from VM
  - Expressiveness vs. Efficiency
    - Some type of events can exact a significant performance penalty
      - Trapping hardware events (interrupts and memory access)
      - Only trapping events that would imply definite misuse
        - » Modification of sensitive memory that should never change at runtime

Design- VMM interface

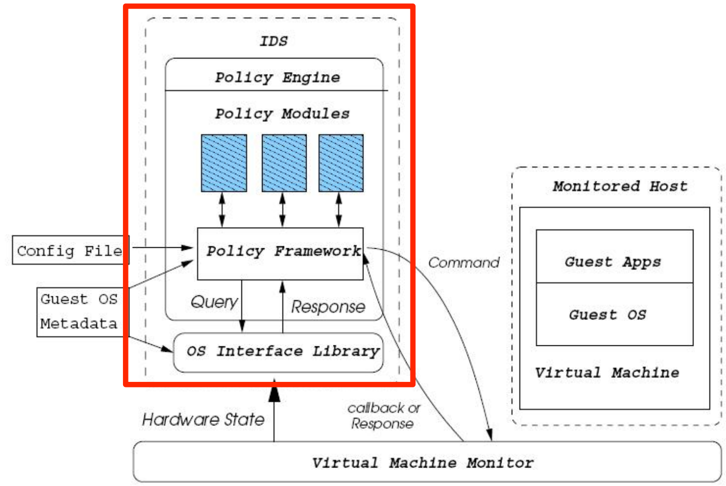


- Communication between VMM and
- Three types of command
  - **Inspection command**
    - Directly examine VM state such as memory and register contents and I/O flags
  - **Monitor command**
    - Events occur and request notification
  - **Administrative command**
    - IDS is allowed to control the execution of a VM
      - Suspend , resume, checkpoint, reboot…

Design- VMI IDS

- Responsible for implementing intrusion detection policies by analyzing machine state and machine events through VMM interface

- Two parts
  - OS Interface Library
  - Policy engine



Design- OS Interface Library

- Provide an OS-level view of the virtual machine's state in order to facilitate easy policy development and implementation
- Consider a situation we want to detect tampering with sshd process
  - VMM can access to any pages of physical memory or disk block in a VM
  - But, "where is virtual memory does sshd's code segment reside?"
- The OS library must be matched with the guest OS

Design- Policy Engine

- Execute IDS policies by using the OS interface library and the VMM interface
- Interpret system state and events from the VMM interface and OS interface library, and decide whether or not the system has been compromised
  - Compromised --> responding in an appropriate manner

Implementation

- Livewire
  - Prototype of VMI IDS
- VMM
  - VMware Workstation for Linux x86
- OS library
  - Mission Critical's crash program (?
- Policy engine
  - Framework and modules
  - Written in Python

Implementation - VMM

- Add hooks to VMware
  - Inspection of memory, registers, and device state
  - Interposition on certain events
    - Interrupts
    - Updates to device and memory state
- **Direct memory access (DMA)**
  - VMM can read any memory location in the VM
- Interactions with virtual I/O devices
  - Intercepted by VMM and mapped actual hardware device
  - Add hooks to notify when the VM attempted to change this state

Implementation – VMM Interface

- Provides a channel for the VMI IDS processes to communicate with VMware VMM process
  - **Unix domain socket**
    - VMI IDS send commands to and receive responses and event notifications from the VMM
  - **Memory-mapped file**
    - Support efficient access to the physical memory of VM

Implementation- Policy Engine

- **Policy framework**
  - A common API for writing security policies
- **Policy modules**
  - Implement actual security policies

Implementation- Policy Framework

- Allow the policy implementer to interact with the major components of the system
  - **OS interface library**
    - A simple request/response to the module writer for sending commands to the OS interface library
    - Receiving responds that have been marshaled in naïve data formats
    - Tables containing key-value pairs that provide information about the current kernel
  - **VMM interface**
    - Direct access to the VM"s physical address and register state
    - Administrative commands
      - Suspend, restart, checkpoint the VM
  - **Livewire frontend**
    - Bootstrapping the system
    - Starting the OS interface library process
    - Loading policy modules
    - Running policy modules

Implementation- Policy Modules

- 6 sample security policy modules in Livewire
  - **Polling modules**
    - Run periodically
    - Check for signs of an intrusion
    - 50 lines of Python
  - **Event-driven modules**
    - Are triggered by a specific event
      - An attempt to write to sensitive memory
    - 30 lines of code

Policy Modules – polling modules

- Periodically check the system for signs of malicious activity
  - **Lie Detector**
    - Directly inspecting hardware and kernel state
    - By querying the host system through user-level program
    - Detect conflict
  - **User Program Integrity Detector**
    - Detect if a running user-level program has been tempered with by periodically taking a secure hash of the immutable sections of a running program
    - Comparing it to known good hash

Policy Modules – polling modules

- **Signature Detector**
  - Perform a scan of all of host memory for attack signatures
  - False positive
- **Raw Socket Detector**
  - A burglar alarm
  - Detecting the use of raw sockets by user-level programs for the purpose of catching such malicious applications

Policy Modules – Event Driven Policy Modules

- Runs when the VMM detects changes to hardware state
  - Each event-driven checker register all of the events it would like to be notified of with the policy framework
  - At runtime, when on of event occurs, the VMM relays a message to the policy framework
  - Policy framework runs the checker which have registered to receive the event

Policy Modules – Event Driven Policy Modules

- **Memory Access Enforcer**
  - Works on marking the code section, sys_call_table, and other sensitive portions of the kernel as read-only through the VMM
  - If a malicious program tries to modify these sections
    - VM will be halted and the kernel memory protection enforcer notified
- **NIC Access Enforcer**
  - Prevents the Ethernet device entering promiscuous mode, or being configured with a MAC address which has not been pre-specified

Experimental(1/3)

- Environment
  - VM
    - 256MB allocation of physical memory
    - 4GB virtual disk
    - Debian GNU/Linux
  - VMM
    - Modified version VMware Workstation for Linux3.1
    - 1.8GHz Pentium IV laptop
    - 1GB physical memory
    - Debian GNU/Linux

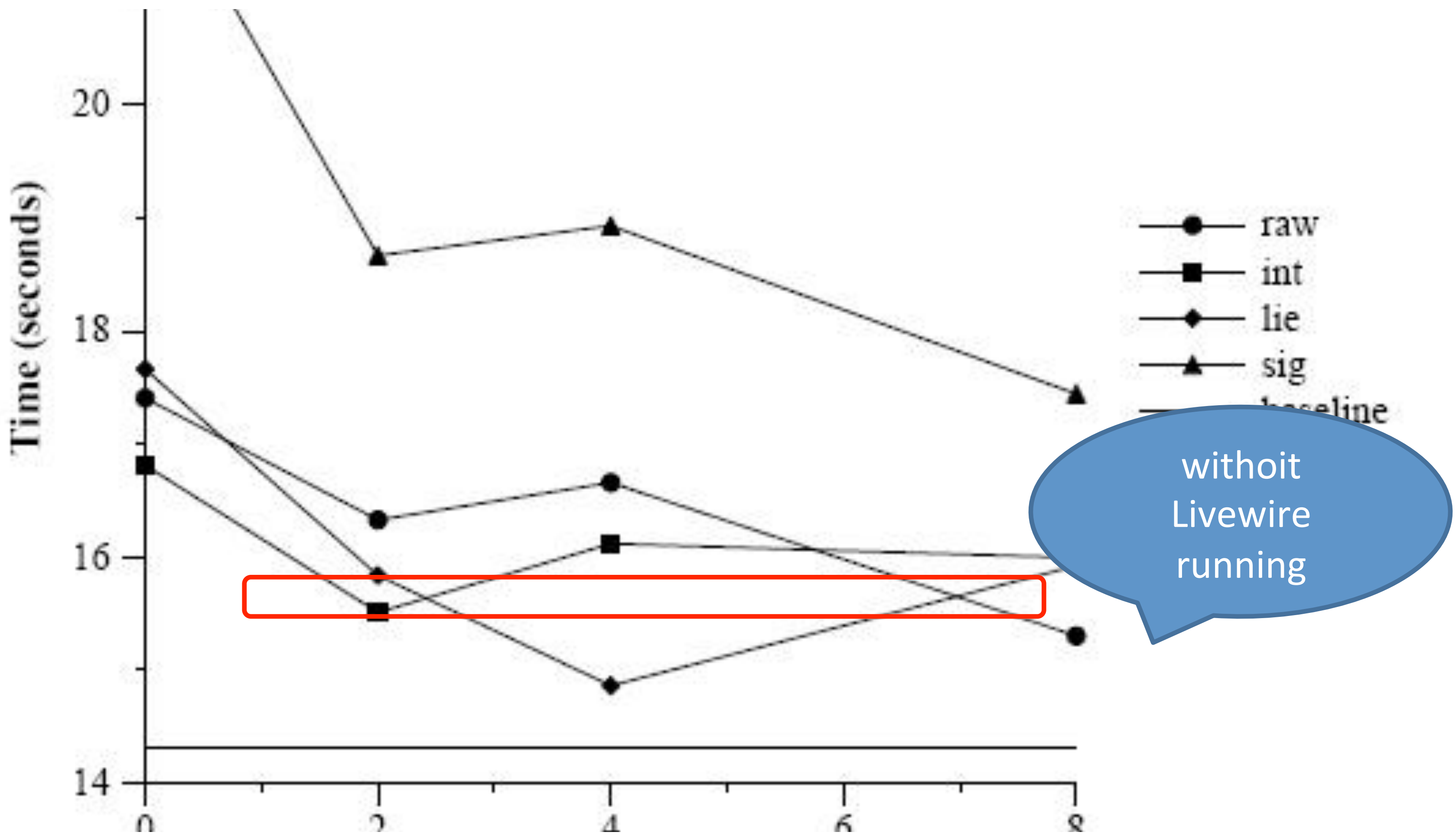
Experimental(2/3) – Detection Result

Description	nic	raw	sig	int
Stealth user level remote backdoor		D		
Precompiled user level rootkit			D	
Linux Worm			D	
Source based user level rootkit	P		D	D
LKM based kernel backdoor/rootkit			D	
LKM based kernel backdoor/rootkit			D	
All-purpose packet sniffer for switched networks	P			
/dev/kmem patching based kernel backdoor			D	

vire policy modules against common attacks. Within the grid, "P"
l attack.

Experimental(3/3) - Performance

- Two work loads
  - Unzipped and untarred the Linux 2.4.18 kernel to provide a CPU-intensive task
    - Evaluate the overhead of running event-driven checkers in the common case when they are not being triggered
    - No measurable overhead
  - Copied the kernel from one directory to another to provide a I/O intensive task

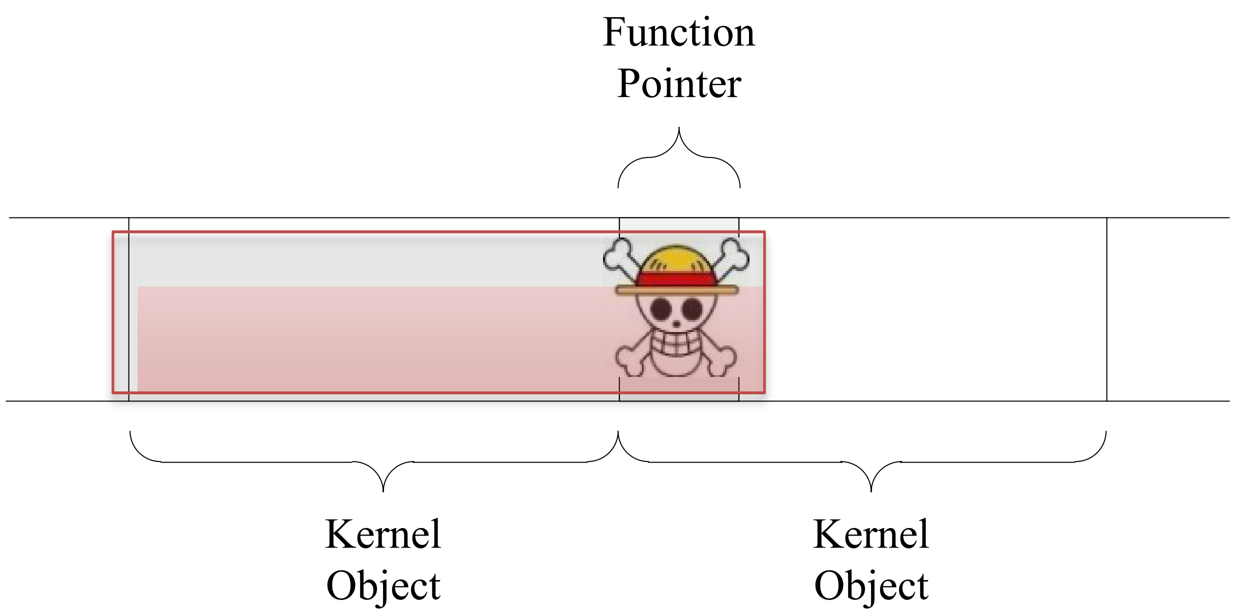

Time (seconds)
20
18
16
14
0
2
4
6
8
raw
int
lie
sig
withoit
Livewire
running

Re-cap of VMI-based IDS

- Propose the idea of VMI IDS
  - High evasion resistance
    - Due to High visibility
  - High attack resistance
    - Strong isolation
  - Detect real attacks with acceptable performance

VMI-based IDS for Kernel-space Buffer Overflow Detection

Kernel Heap Buffer Overflow



Motivation

- An efficient mechanism that detects kernel heap buffer overflows.

Limitations of Current Methods(1/2)

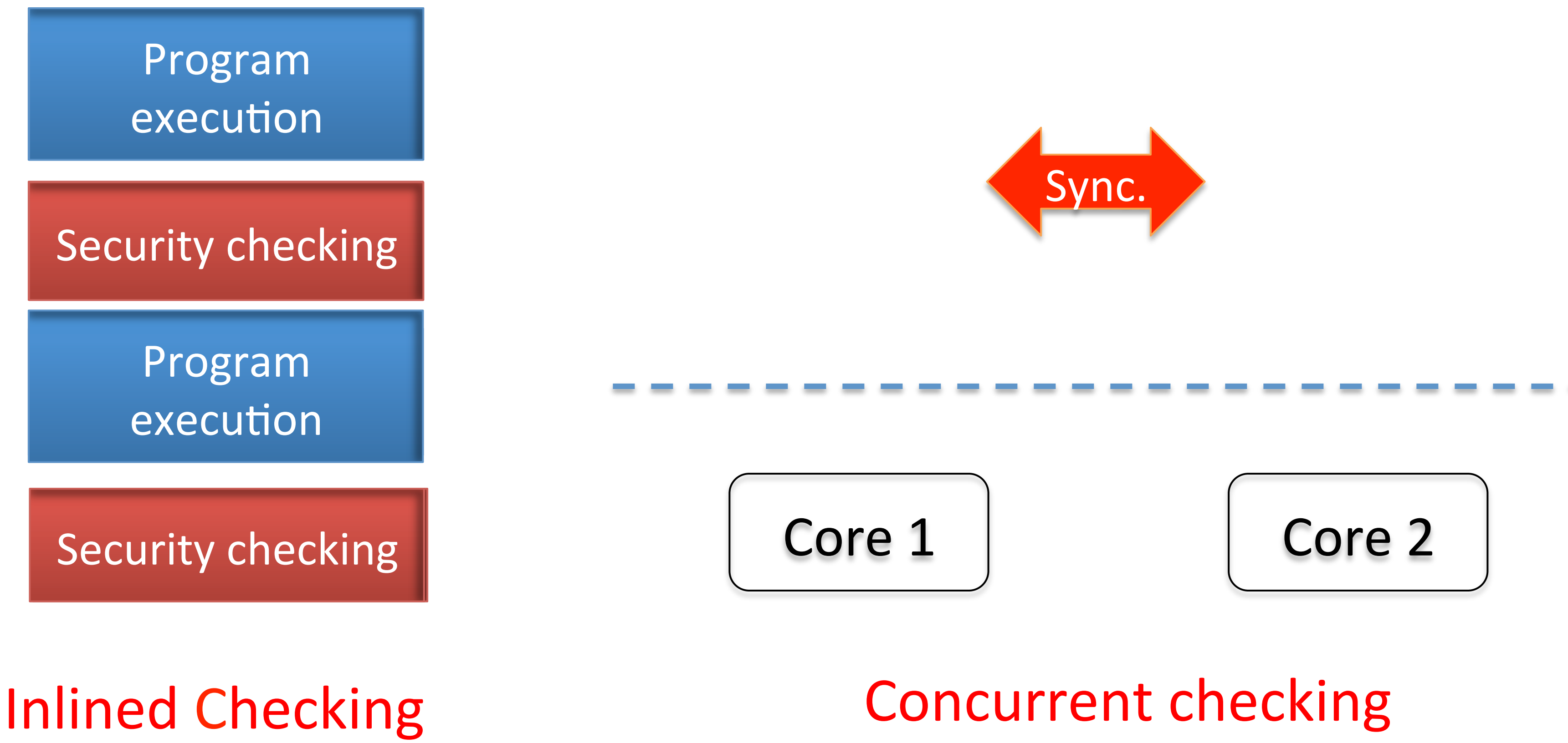
- Some approaches perform detection before each buffer write operation.
  [PLDI '04], [USENIX ATC '02], [NDSS '04]

- Some approaches do not check heap buffer overflows until a buffer is de-allocated.
  [LISA '03], [BLACKHAT '11]

Limitations of Current Methods(2/2)

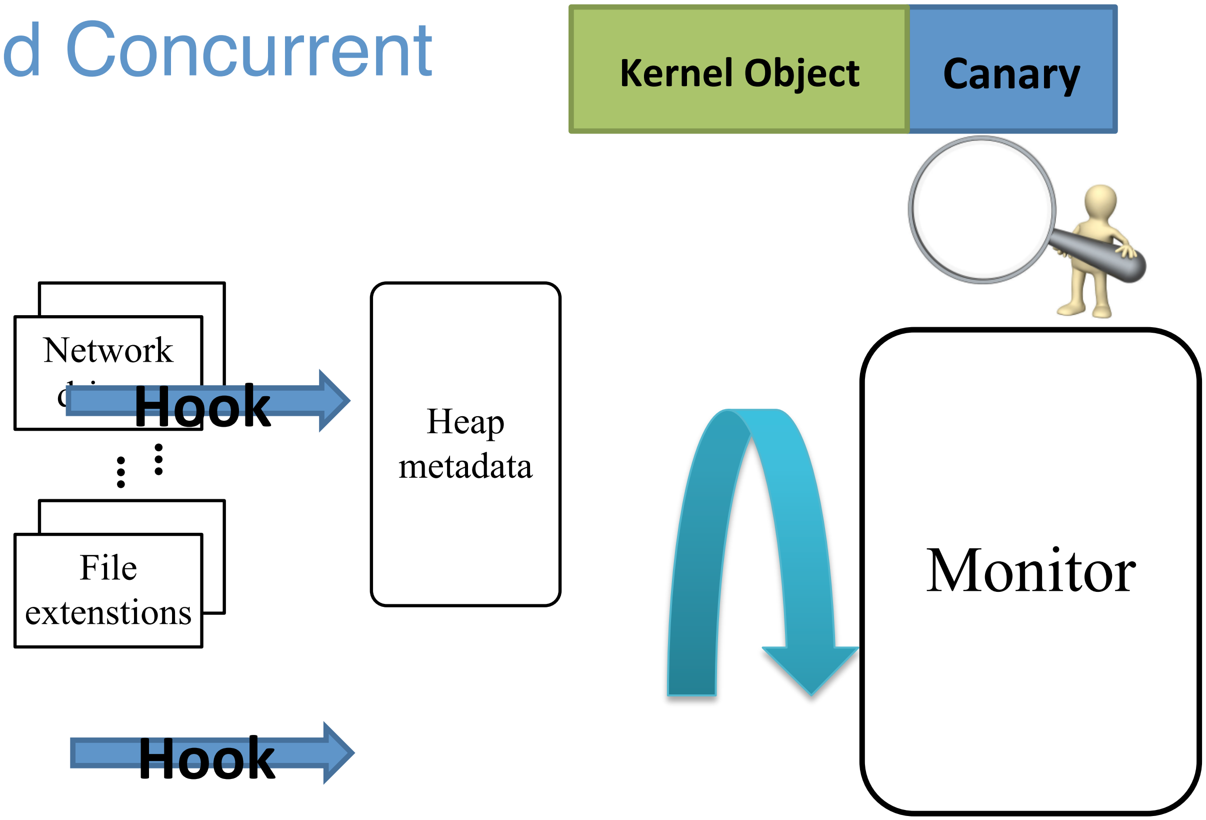
- Some approaches either rely on special hardware or require the operating system to be ported to a new architecture.
  [USENIX Security '08], [EuroSys '09]

Our Idea


Program execution
Security checking
Program execution
Security checking
Inlined Checking
Sync.
Core 1
Core 2
Concurrent checking

Basic Method

- Canary-based Concurrent Monitoring

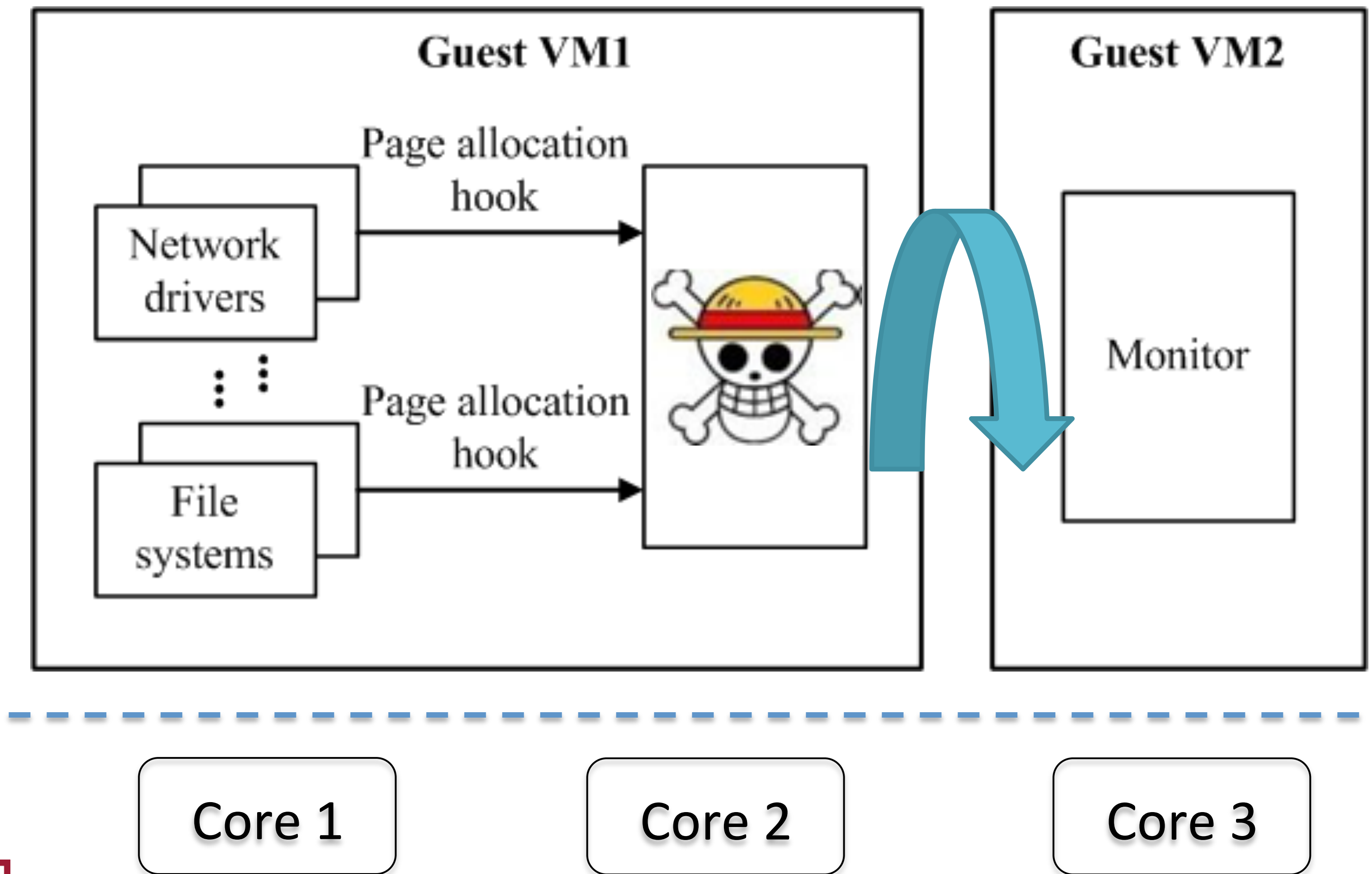


Challenges

- Synchronization.
  - Sharing kernel heap metadata
- Self-protection.
  - Monitor and the metadata
- Compatibility.
  - OS and hardware

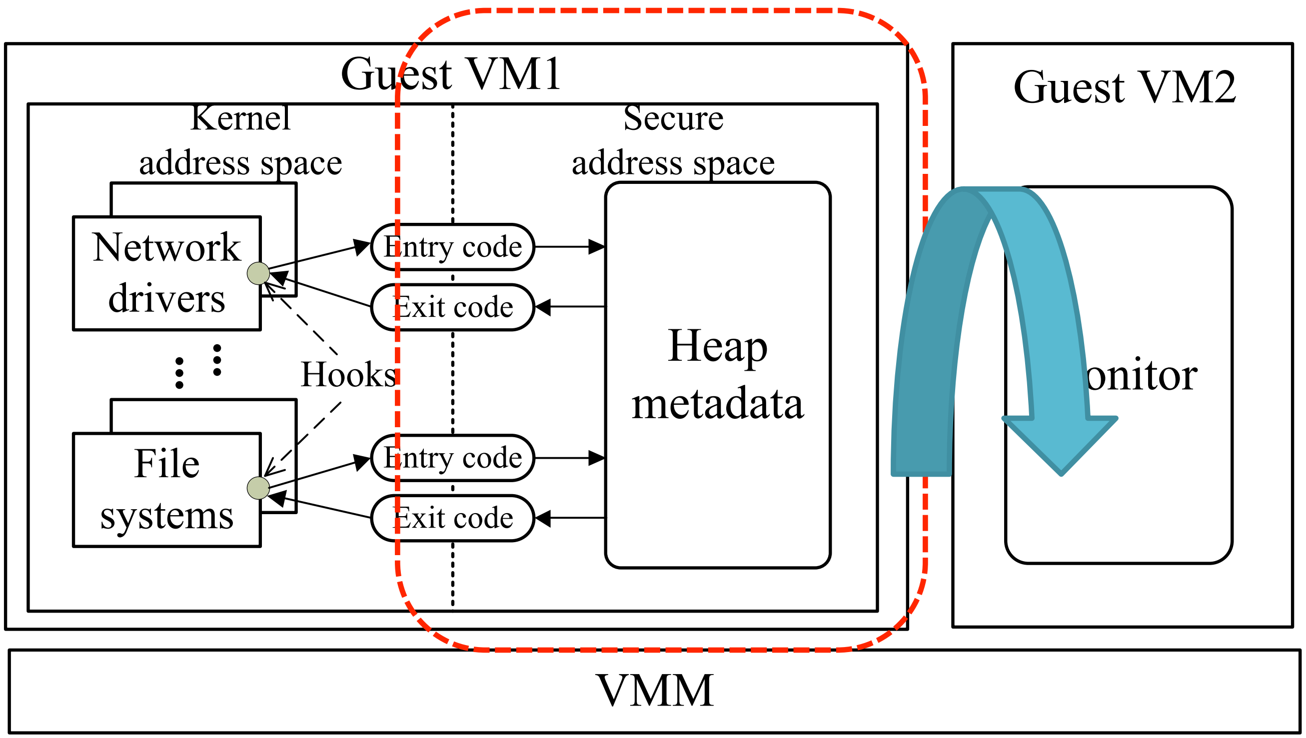
Out-of-the-VM Architecture

(*Our previous CCS submission - rejected*)



Hybrid VM monitoring Architecture

(*NDSS submission - accepted*)



Now, Kernel Cruising

- Metadata

- Races between target kernels and monitor

Kernel Cruising

- Page Identity Array (PIA)
  - Heap buffer canary location information
  - Other information

- Race conditions
  - Non-atomic entry write
  - Non-atomic entry read
  - Time of check to time of use

Semi-synchronized Non-blocking Cruising Algorithm

- Avoid Concurrent Entry Updates.
  - Put the PIA entry update operations into the critical section.
  - **Update the flag**.
- Identify Time of check to time of use.
  - Use a double-check algorithm (**with the flag**) to detect potential inconsistency.
- **Using the flag may cause ABA hazards!**

ABA hazard example

if the page is moved to the heap page pool

flag = true;

else if the page is removed from the heap

flag = false;

…

if (the canary is

true->false->true

A B A

if (flag == true

the page is still used by the original slab

}

}

ABA Hazard Solution

if the page is moved to the heap page pool

version++;

else if the page is removed from the heap

version++;

…

if (the canary is tempered) {

*if (**version == original version**) {*

the page is still used by the original slab

}

}

Non-blocking Cruising Algorithm

```
Monitor(){
uint ver1, ver2;
for (int page = 0; page < ENTRY NUMBER; page++){
    ver1 = PIA[page].version;
    if (The page is non-heap page)
        continue; // Bypass non−heap page
    Read the metadata stored in PIA[page];
    ver2 = PIA[page].version;
    if (ver1 != ver2)
        continue; // Metadata was updated
    for (each canary within the page){
        if (the canary is tampered){
            DoubleCheckOnTamper(page, ver1);
        }
    }
}
```

Avoid Read Inconsistency!

Is the page still used by the heap?

Secure Canary Generation

- R1) Attackers cannot recover the corrupted canaries after the kernel is compromised.
- R2) The canary generation and verification algorithms should be efficient.
- Generate unpredictable canaries using RC4 from a per-virtual-page random value.

Guaranteed Detection

- The In-VM protection prevent attackers from manipulating the PIA entries.
- The canary cannot be predictable thanks to the stream cipher.

Outline

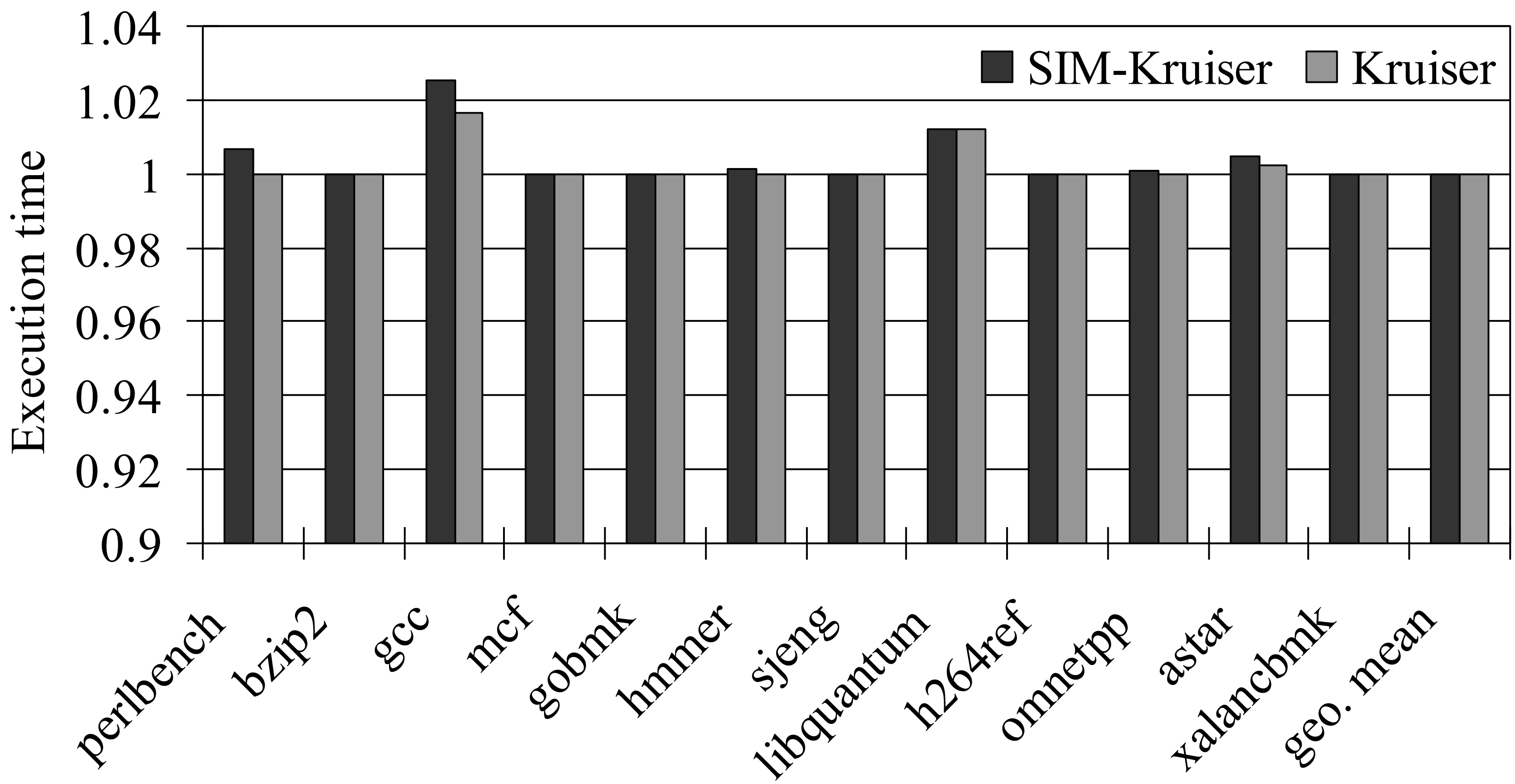
- Idea
- Architecture
- Kernel Cruising
- **Evaluation**
- Related Work
- Summary

Effectiveness

- We exploited five heap buffer overflow vulnerabilities in Linux, including three synthetic bugs and two real world vulnerabilities .

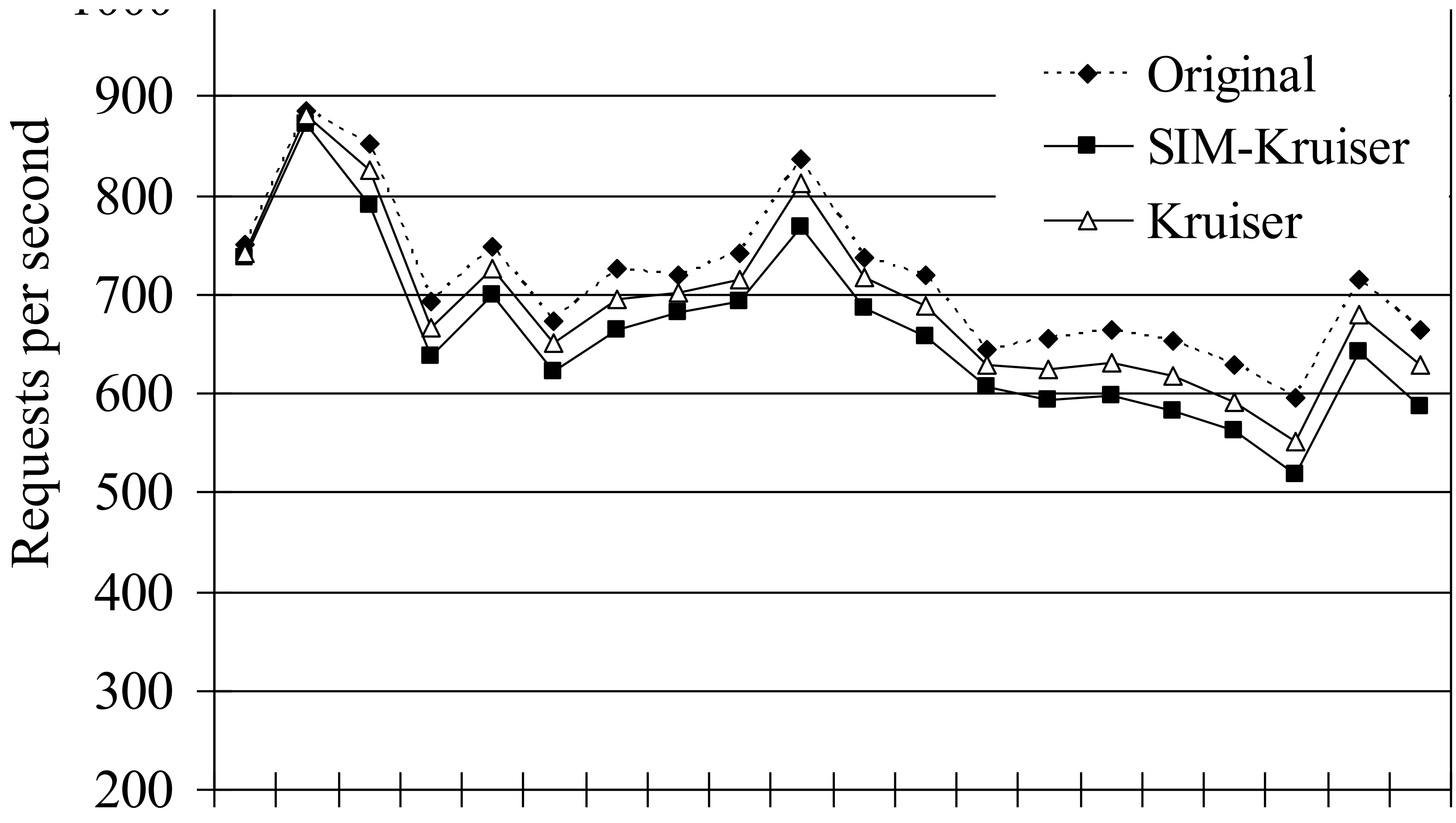
- All the overflows are successfully detected by *Kruiser*.

Performance Overhead



SPEC CPU2006 performance (normalized to the execution time of original Linux).

Scalability



Throughput of the Apache web server for varying numbers of concurrent requests.

Detection Latency

Different cruising cycle for different applications in the SPEC CPU2006 benchmark

mark	Maximum cruising number	Minimum cruising number	Average cruising number	Averag cruising cy
ench	107,824	105,145	106,378	39,25
2	79,085	76,325	76,682	27,66
c	78,460	76,810	77,413	27,77
f	82,885	79,328	79,540	28,15
nk	80,761	80,345	80,519	28,60
ıer	81,278	80,435	80,591	28,63
ıg	81,437	80,259	80,535	28,61
ntum	80,911	80,317	80,407	28,49
ref	80,756	80,337	80,480	28,57
tpp	82,109	80,796	81,088	28,83
ır	81,592	81,022	81,097	28,89
bmk	99,436	82,747	88,454	30,19

Outline

- Idea
- Architecture
- Kernel Cruising
- Evaluation
- **Related work**
- Summary

Related Work

- Countermeasures Against Buffer Overflows
  - StackGuard [USENIX Security '98]
  - Heap Integrity Detection [LISA '03]
  - Cruiser [PLDI '11]
  - DieHard [PLDI '06] and DieHarder [CCS '10]
- VM-based Methods
  - SIM [CCS '09]
  - OSck [ASPLOS '11]

Summary

- *Kruiser* can achieve *concurrent monitoring* against kernel heap buffer overflows.
  - *Non-blocking*
  - *Semi-synchronized*
  - *NO false positive*

- The *hybrid VM monitoring* scheme provides high efficiency without sacrificing the security guarantees.