Architectural Supports to Protect OS Kernels from Code-Injection Attacks

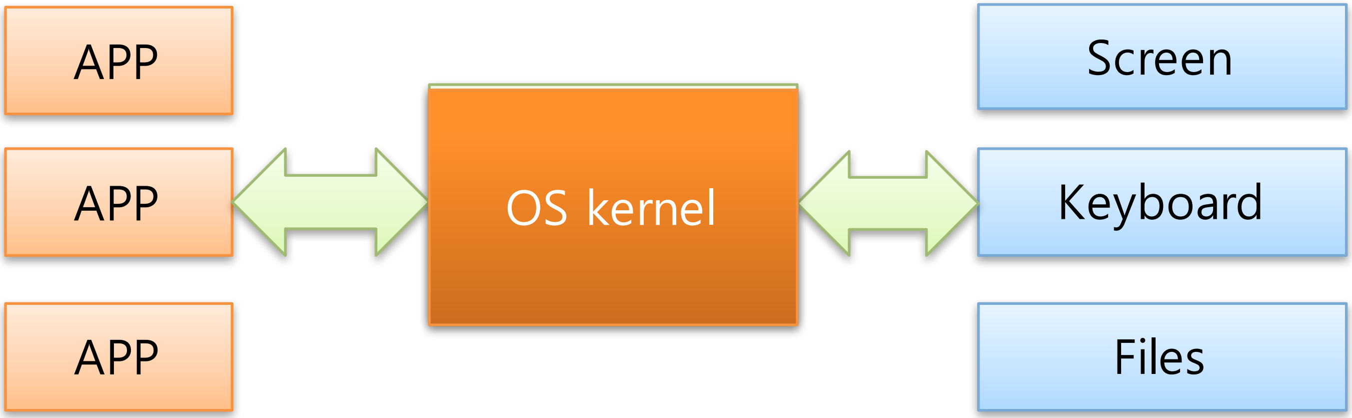
2016-06-18

Hyungon Moon, Jinyong Lee, Dongil Hwang,
Seonhwa Jung, Jiwon Seo and Yunheung Paek
Seoul National University

Why to protect the OS kernels?

- Operating systems (and their kernels) are everywhere

- Applications rely on the OS kernels



APP

APP

APP

OS kernel

Screen

Keyboard

Files

Operating systems are vulnerable

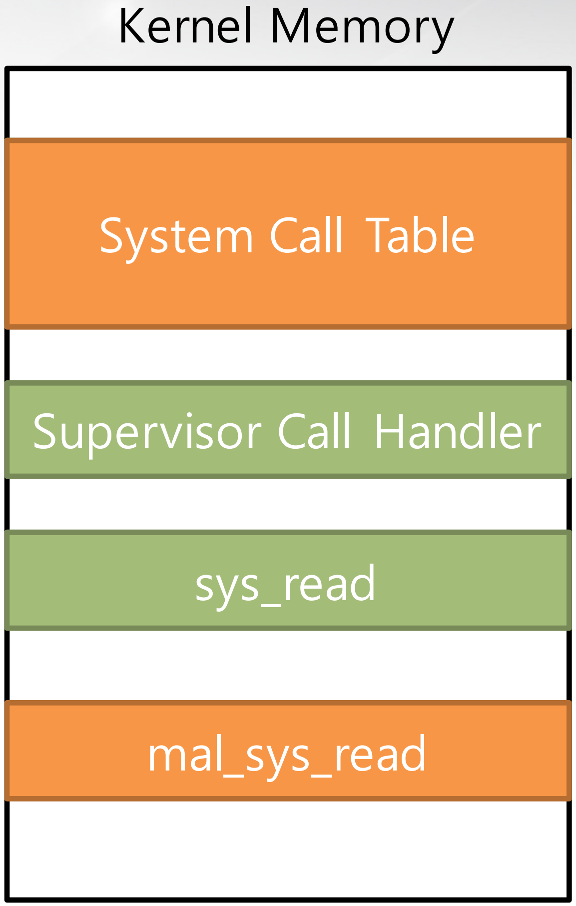
- New vulnerabilities reported every year
  - CVE-2013-2094 (S. Vogl et al., 2014)
  - CVE-2014-3153 (TowelRoot)
  - CVE-2015-3636 (PingPongRoot)

- Adversaries may
  - **Read** from the memory regions for the kernel
  - **Write** to the memory regions for the kernel

- With the capabilities,
  - Hiding Processes, files, or network connections
  - Privilege escalation
  - **Execute their code while the CPU in the kernel mode**

A powerful type of attack: code-injection

- Handling a read system call
  - Supervisor call handler
    → sys_read
  - The address of sys_read written in the system call table
- Attackers can
  - Write their code into the kernel's memory
  - Manipulate the system call table
- Consequence
  - mal_sys_read replaces sys_read



Kernel Memory

System Call Table

Supervisor Call Handler

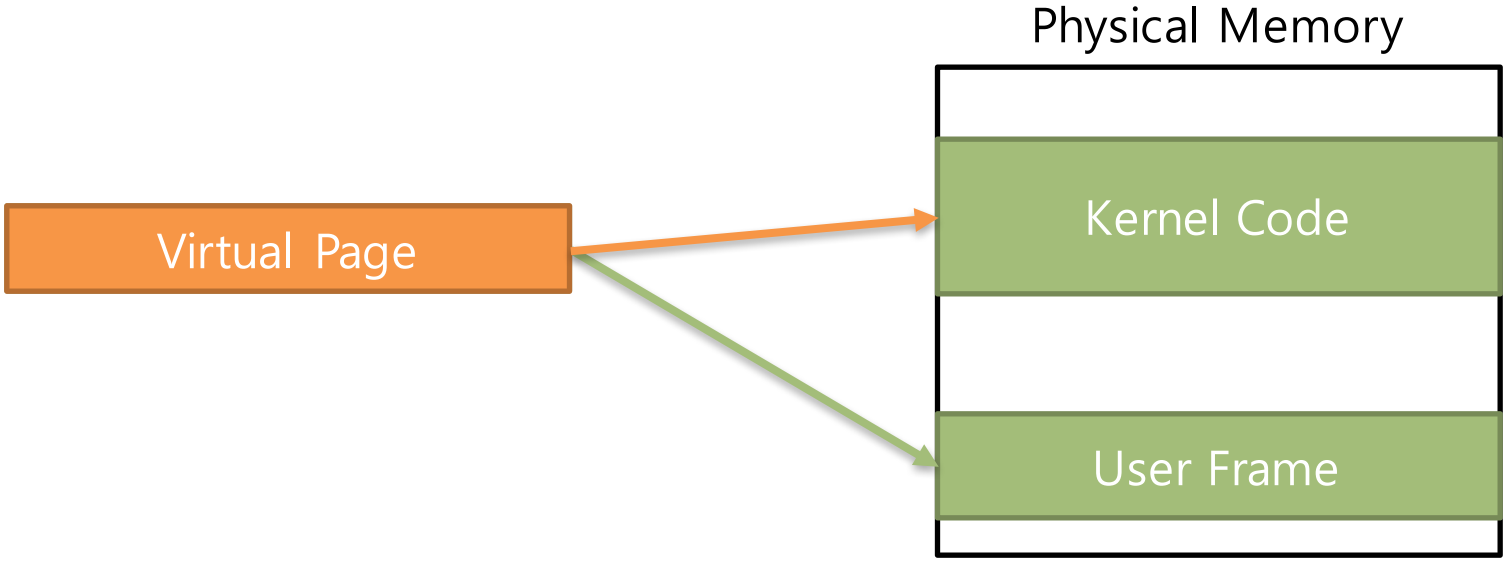
sys_read

mal_sys_read

Existing mechanisms effective

- Privileged eXecute Never (PXN)
  - A flag in the page table entries
  - MMU prevents the execution of memory pages with PXN=1

- Page Table Protection ⇒ No Code-Injection Attack



Kargos overview

Goal

- Mitigate the kernel code-injection attacks with minimal performance cost

Threat Model

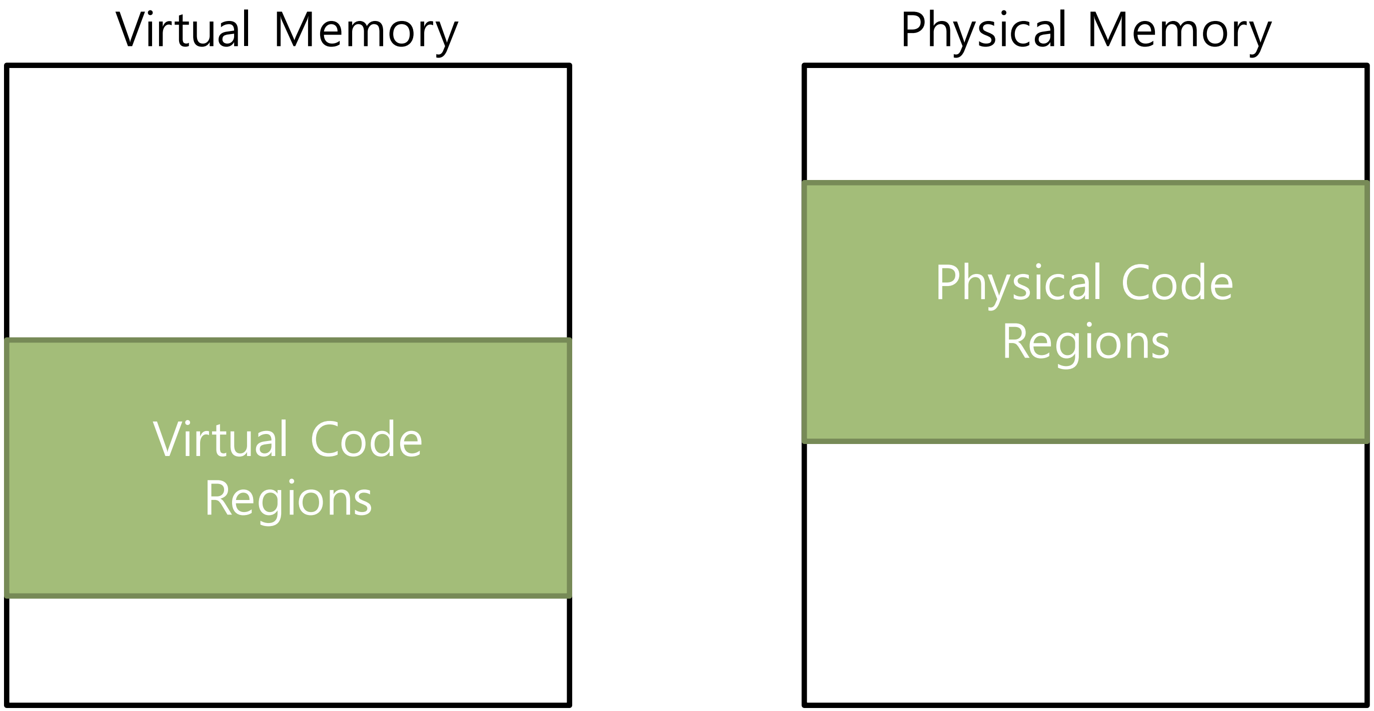
- Adversaries can **read** from/**write** to the kernel memory arbitrarily

Mechanism

- Dedicated hardware support
  - Traffic Monitor
  - Trace Monitor
- Minimal kernel instrumentation
  - Special execution traces
  - Special register protection

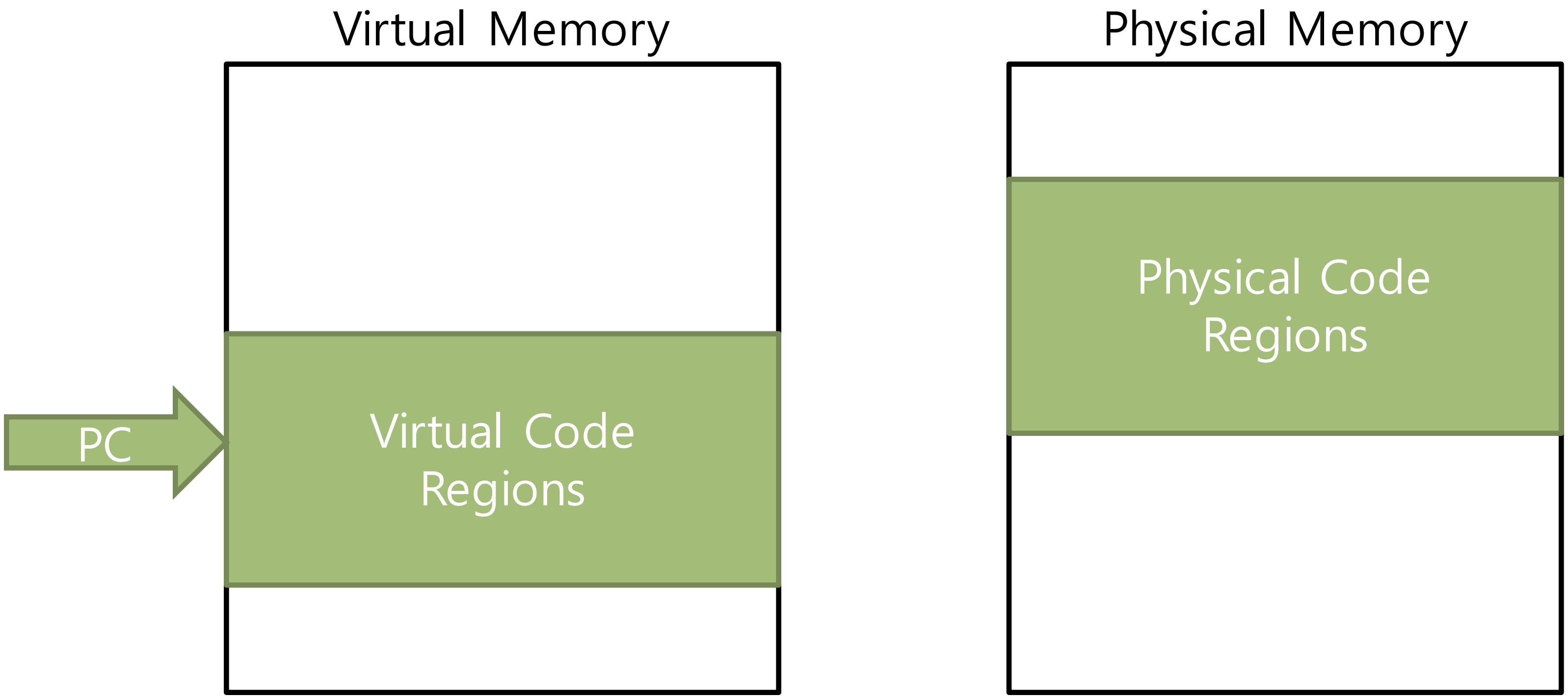
The four rules to detect the attacks

R1. The physical code regions of the kernel should never be modified



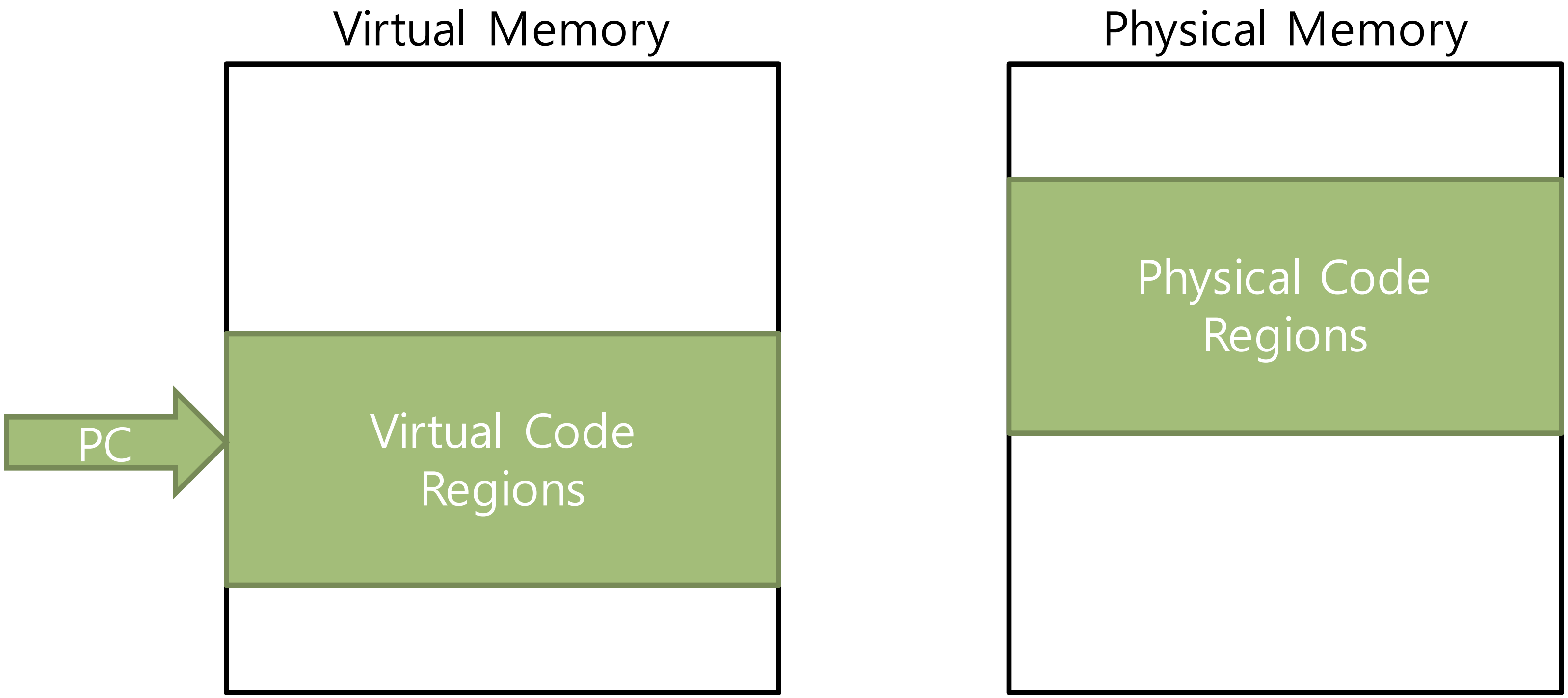
The four rules to detect the attacks

R2. The CPU jumps to an address in the virtual code regions when entering the kernel



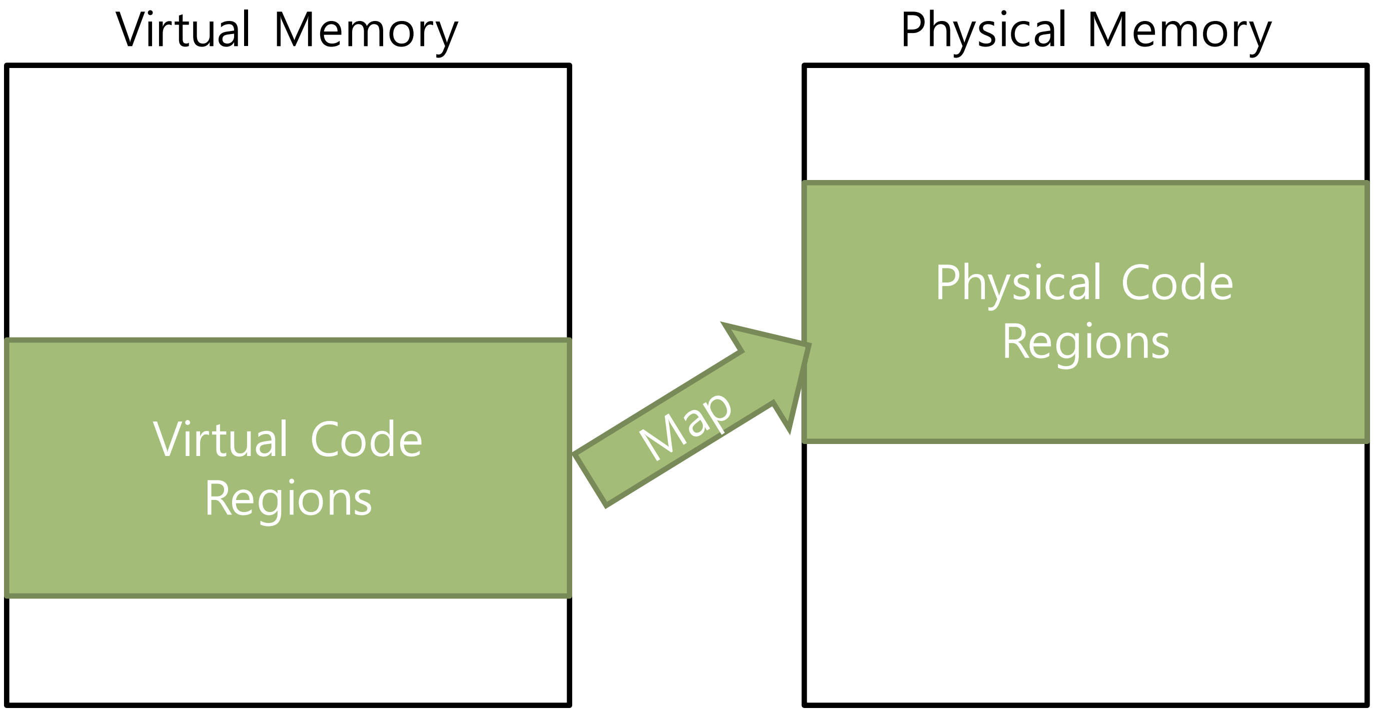
The four rules to detect the attacks

R3. All indirect branch targets lie in the virtual code regions while the CPU is in the kernel mode



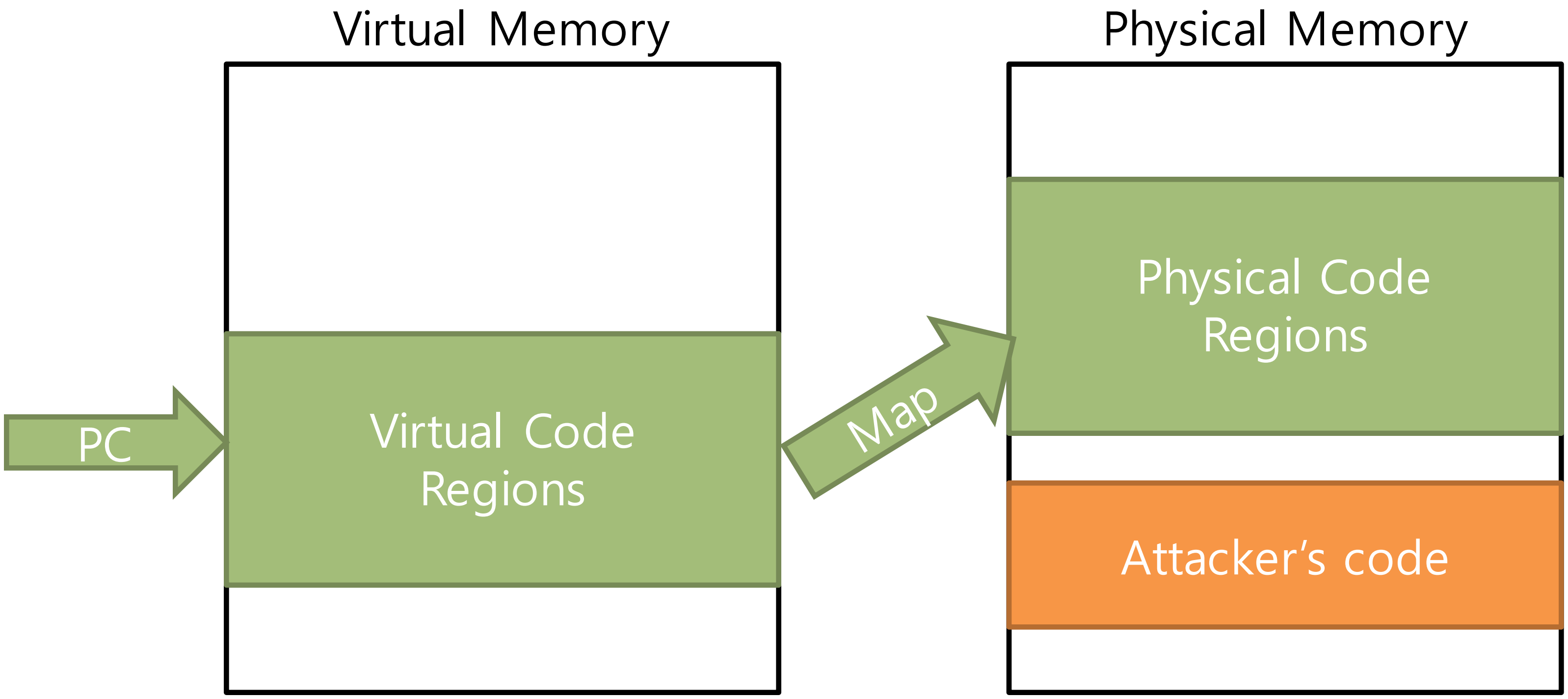
The four rules to detect the attacks

R4. All virtual code regions are mapped to the physical code regions.



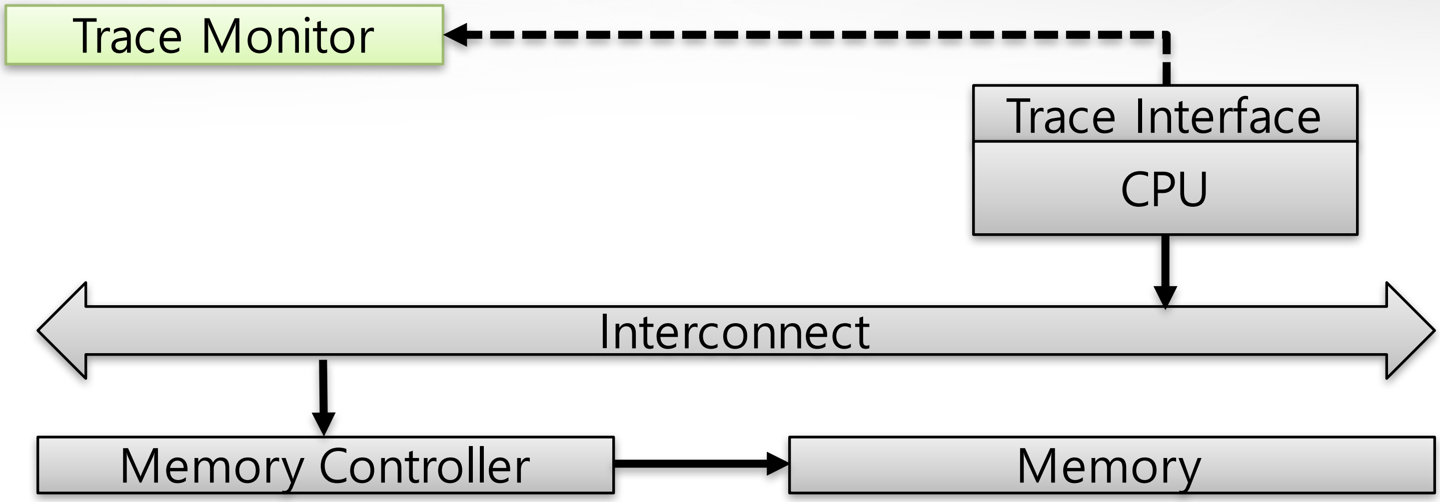
Why the four rules prevent the attacks

- R1: attacker's code should be outside the physical regions
- R2 & R3: PC points to the virtual code regions
- R4: Virtual code regions never mapped to the attacker's code



Trace monitoring

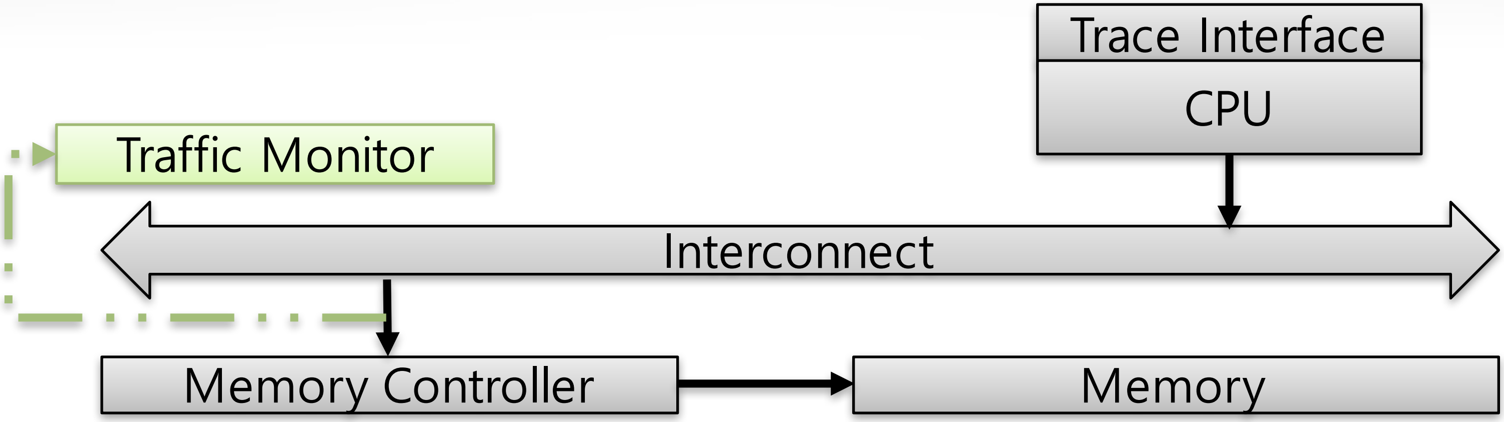
- Need to monitor the virtual addresses that the CPU jumps to



- Our Implementation:
  - Parses the ARM's PTM packets

Traffic monitoring

- Need to know the physical addresses that the CPU writes to

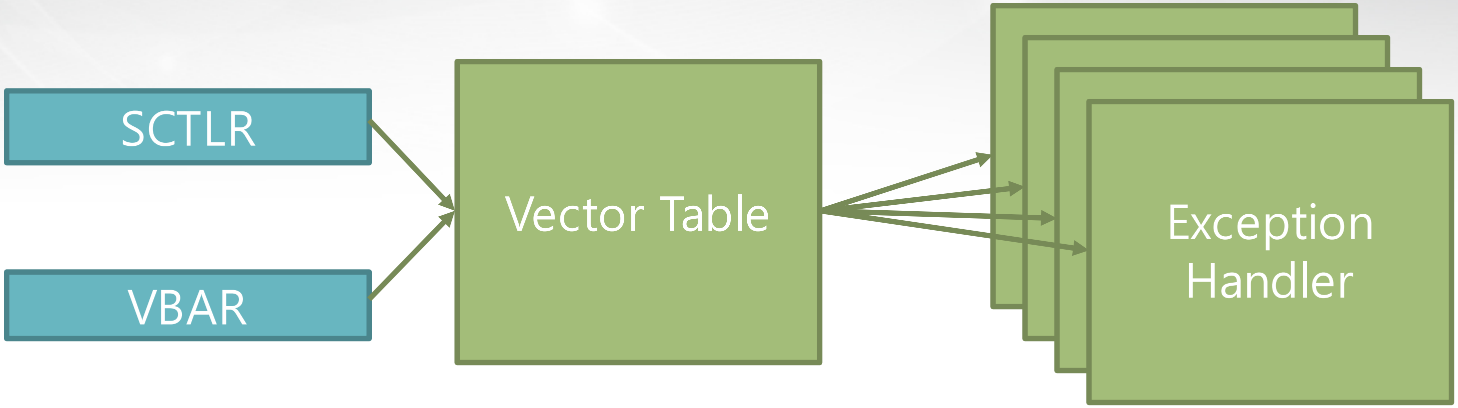


- Our implementation:
  - Examines the traffic complying with the AXI protocol

- Naturally detect the violations of R1

Rule 2: Kernel entrance

- The gateway code blocks



- Vector table is inside the physical code regions
- Protection of the SCTLR and VBAR: Kernel Instrumentation
  - Check the values before executing the special instructions

Rule 3: Indirect branches

- Challenge: Mode recognition
  - In which CPU mode a trace is generated?
  - Jump to gateway code block indicates the kernel enter
- Answer: special traces in the exit code blocks

```
msr       SPSR_fsxc, r1
and       r3, r1, #31
cmp       r3, #16
subeq     pc, pc, #4
restore_context
movs      pc, lr
```

Rule 4: Mappings

- Memory management unit uses:

- Partial page table protection
  - Small number of (<10) PGD entries for virtual code region translations
  - Traffic Monitor can detect the modifications
- TTBR protection: Kernel Instrumentation
  - Check the PGD entries before updating the TTBRs

Prototype implementation details

- Implemented all hardware components in Verilog HDL
- Used Xilinx ZC702 evaluation kit to prototype
- Operational frequency:
    - Processor core: 222MHz
    - Kargos hardware modules: 80MHz

- Kernel instrumentations
    - Six for SCTLR updates
    - Four for TTBR updates
    - Two exit code blocks

Evaluation: Security

- Implemented three Proof-of-Concept(PoC) attacks using a real-world vulnerability (CVE-2014-3153)
  - Kernel code modification
  - Virtual code region remapping
  - Redirecting the kernel execution to a attacker's code block
- Targeting Linux kernel 3.8.0 for Android 4.2
- All these three attacks detected

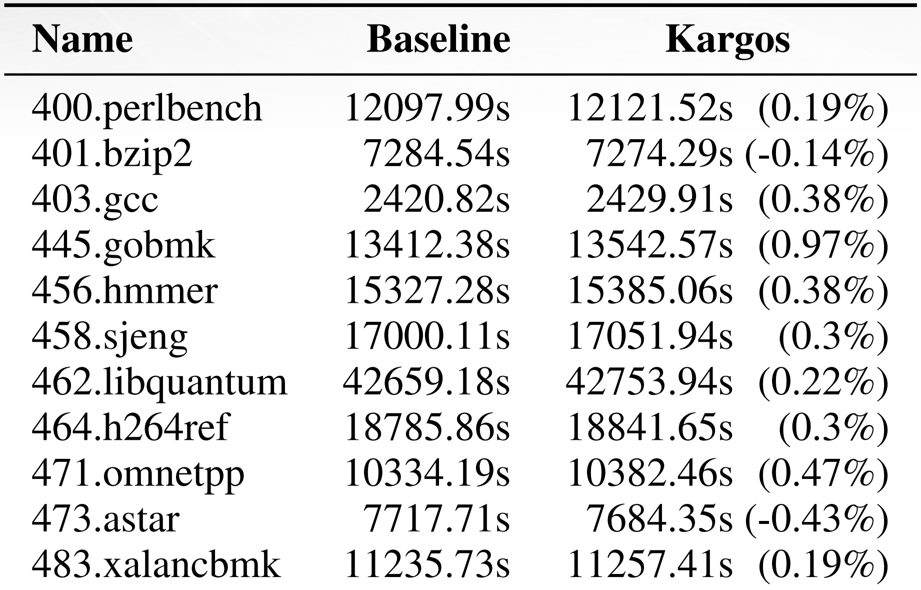
Evaluation: Performance 1

- LMBench result to show the impact on OS services

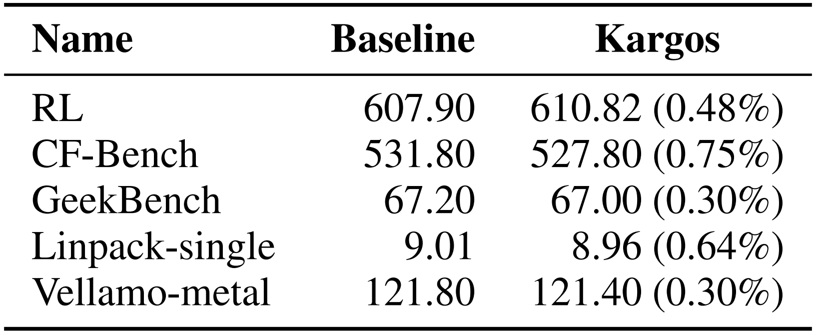
Name	Baseline	Kargos
null syscall	0.98μs	1.07μs (0.92%)
open/close	18.39μs	18.15μs (-1.28%)
select	4.58μs	4.57μs (-0.11%)
sig. handler install	2.81μs	2.82μs (0.11%)
sig. handler overhead	9.91μs	10.55μs (6.42%)
pipe	40.89μs	43.23μs (5.72%)
fork+exit	2853.15μs	2838.60μs (-0.51%)
fork+execve	9279.8μs	9159.16μs (-1.3%)
page fault	4.34μs	4.45μs (3.63%)
mmap	84.7μs	84.9μs (0.24%)

Evaluation: Performance 2

- Application benchmarks for the comparison



Name	**Baseline**	**Kargos**
400.perlbench	12097.99s	12121.52s (0.19%)
401.bzip2	7284.54s	7274.29s (-0.14%)
403.gcc	2420.82s	2429.91s (0.38%)
445.gobmk	13412.38s	13542.57s (0.97%)
456.hmmer	15327.28s	15385.06s (0.38%)
458.sjeng	17000.11s	17051.94s (0.3%)
462.libquantum	42659.18s	42753.94s (0.22%)
464.h264ref	18785.86s	18841.65s (0.3%)
471.omnetpp	10334.19s	10382.46s (0.47%)
473.astar	7717.71s	7684.35s (-0.43%)
483.xalancbmk	11235.73s	11257.41s (0.19%)



Name	**Baseline**	**Kargos**
RL	607.90	610.82 (0.48%)
CF-Bench	531.80	527.80 (0.75%)
GeekBench	67.20	67.00 (0.30%)
Linpack-single	9.01	8.96 (0.64%)
Vellamo-metal	121.80	121.40 (0.30%)

Conclusion

- Detection of kernel code injection attacks is not expensive
  - With appropriate hardware supports

- Hardware monitors can examine CPU states
  - Mode of execution (privileged/user)
  - Special register values

- Can this mechanism also applied for the detection of the code-reuse attacks?

NLRL

National Leading Research Lab.

미래창조과학부

Thank you!