NDSS 2011

Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions

Xi Xiong, Donghai Tian and Peng Liu
The Pennsylvania State University

PENNSTATE 1855

Outline

- **Motivation**
- Approach Overview
- Key Design & Implementation
- Evaluation
- Summary

Background

- Kernel compromise through extension interface
  - Malware: kernel-level rootkits
    - e.g., subvert kernel meta data or control flow to hide malicious activities
  - Buggy extensions
    - Linux drivers are seven times more likely to contain bugs than other kernel code. [Chou, SOSP 01]
  - Malicious Device Drivers

Related Work

- Prohibit execution of untrusted code
  - Secvisor [Seshadri '07], NICKLE [Riley '08]...
- Kernel control data protection
  - HookSafe [Wang '09]...
- Monitor the behavior
  - K-Tracer [Lanzi `09], Poker [Riley `09]...
- Find signatures and invariants
  - Gibraltar [Baliga '08], Robust Signature [Dolan-Gavitt '09], KOP [Carbone '09], SigGraph [Lin '11]...
- Our approach: shepherd untrusted extensions

Problem we focus on

- How to let untrusted kernel extensions safely run to provide desired functionalities without harming the integrity of the OS kernel?

Outline

- Motivation
- **Approach Overview**
- Key Design & Implementation
- Evaluation
- Summary

Kernel Integrity Threatened by extensions

- Kernel Code/Data Integrity
- Architectural state integrity
- Control flow integrity
  - e.g., extensions jump to undesired positions of kernel text
- Stack integrity
  - e.g., inject malicious kernel stack frames

Basic idea…

- Using run-time access control to limit (shepherd) what untrusted extensions can do.
- examples:
  - untrusted extensions cannot change the kernel code
  - they cannot write to high integrity data objects owned by kernel, but kernel can
  - they can only invoke a limited set of kernel APIs
  - they can only write to its own stack frames

Practical Challenges I

- In commodity OS, extensions and OS kernel are in the same execution context (no context switch)
  - subject identification: who is running? extension or kernel?
- Kernel and extension are in the same address space with less meta information
  - object identification: figure out which part of physical memory contains which type of objects.

Practical Challenges II

- Writing to kernel objects are directly through memory operations, no existing interface to place authorization hooks
  - system calls, LSM
  - mediation and enforcement challenge

- How to monitor control flow transfer and guarantee its integrity?

Approach Overview

- HUKO: a hypervisor based protection system
  - mediation on kernel-extension interaction
  - run-time mandatory access control
- Overview

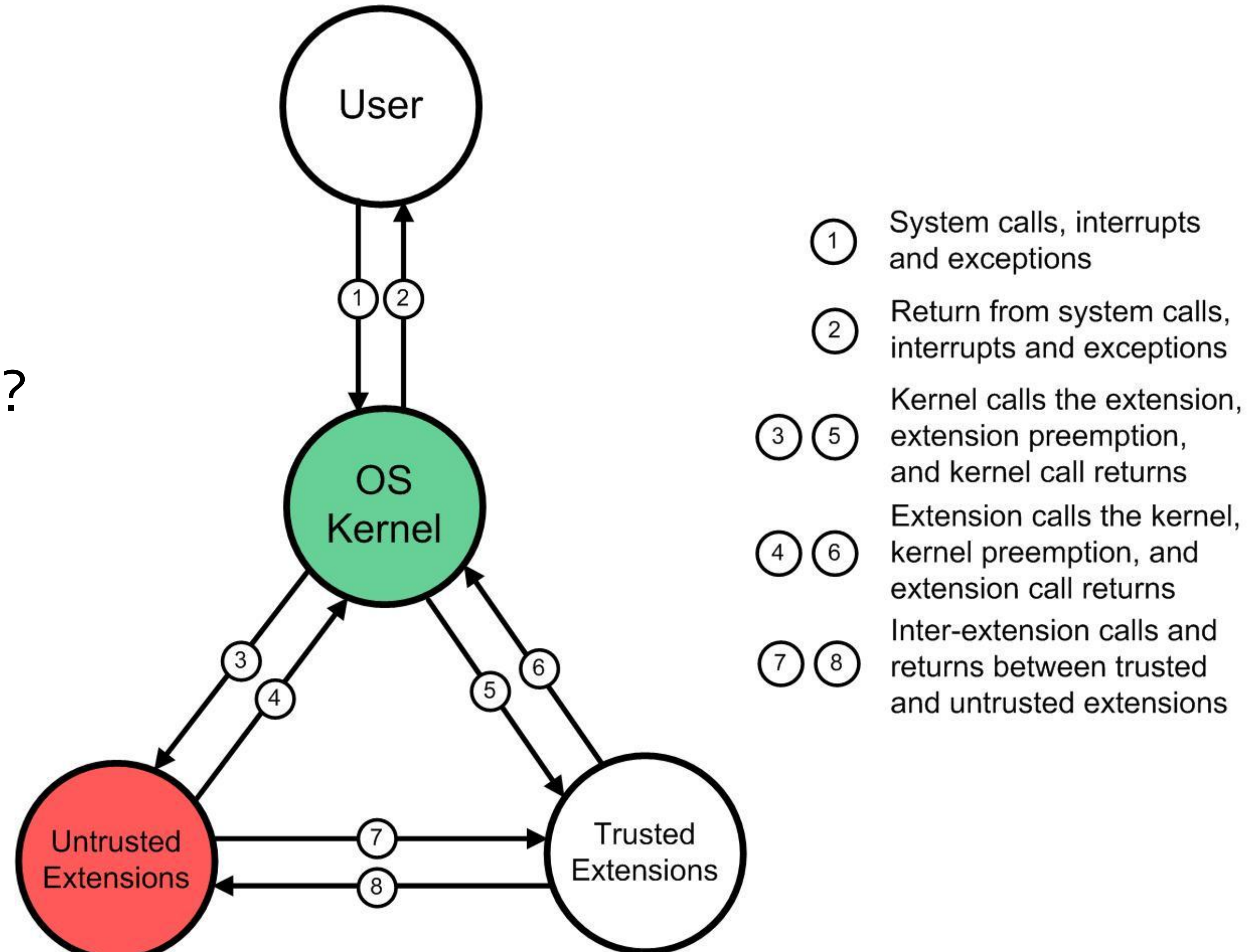
Challenge	Design Solution
Subject Identification	Protection States
Object Identification	Page-based kernel object labeling
Mediation and Enforcement	VMM-level protection domains using Hardware assisted paging
Control Flow Integrity	Trusted Entry Points, call-return consistency

Outline

- Motivation
- Approach Overview
- **Key Design & Implementation**
- Evaluation
- Summary

Protection States

- Who is running?



Object Labeling

- Type-based labeling
  - e.g., KERNEL_CODE, KERNEL_DATA, UNTRUSTED_CODE
- Labels are associated with corresponding physical pages
- Need assistance from OS for
  - extension loading
  - dynamic page allocation and reclaiming
- Issue: Mixed pages
  - Code and data, Trusted and untrusted content, superpages

Access control policy

Object Label	Subject Category / Protection State								
	OS Kernel			Trusted Extensions			Untrusted Extensions		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
Trusted Entry Points	allow	allow	allow	allow	allow	audit allow	allow	deny	audit allow
Other OS Code	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
OS Data	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
Trusted Extension	allow	allow	audit allow	allow	allow	allow	allow	deny	deny
Untrusted Extension	allow	allow	audit allow	allow	allow	audit allow	allow	allow	allow
Private Stack Frames	allow	allow	deny	allow	allow	deny	allow	allow	deny
Other Stack Frames	allow	allow	deny	allow	allow	deny	allow	deny	deny
Trusted DMA	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
Shared DMA	allow	allow	allow	allow	allow	allow	allow	allow	allow
User Space Content	allow	allow	audit allow	allow	allow	audit allow	allow	allow	deny

Memory Isolation

- Basic idea: create hardware enforced protection domains
  - address space separation
  - protection state transition: implemented by domain switch

- How to achieve?
  - **multiple** sets of page tables for different protection domains, **switch** the page table upon protection state transition

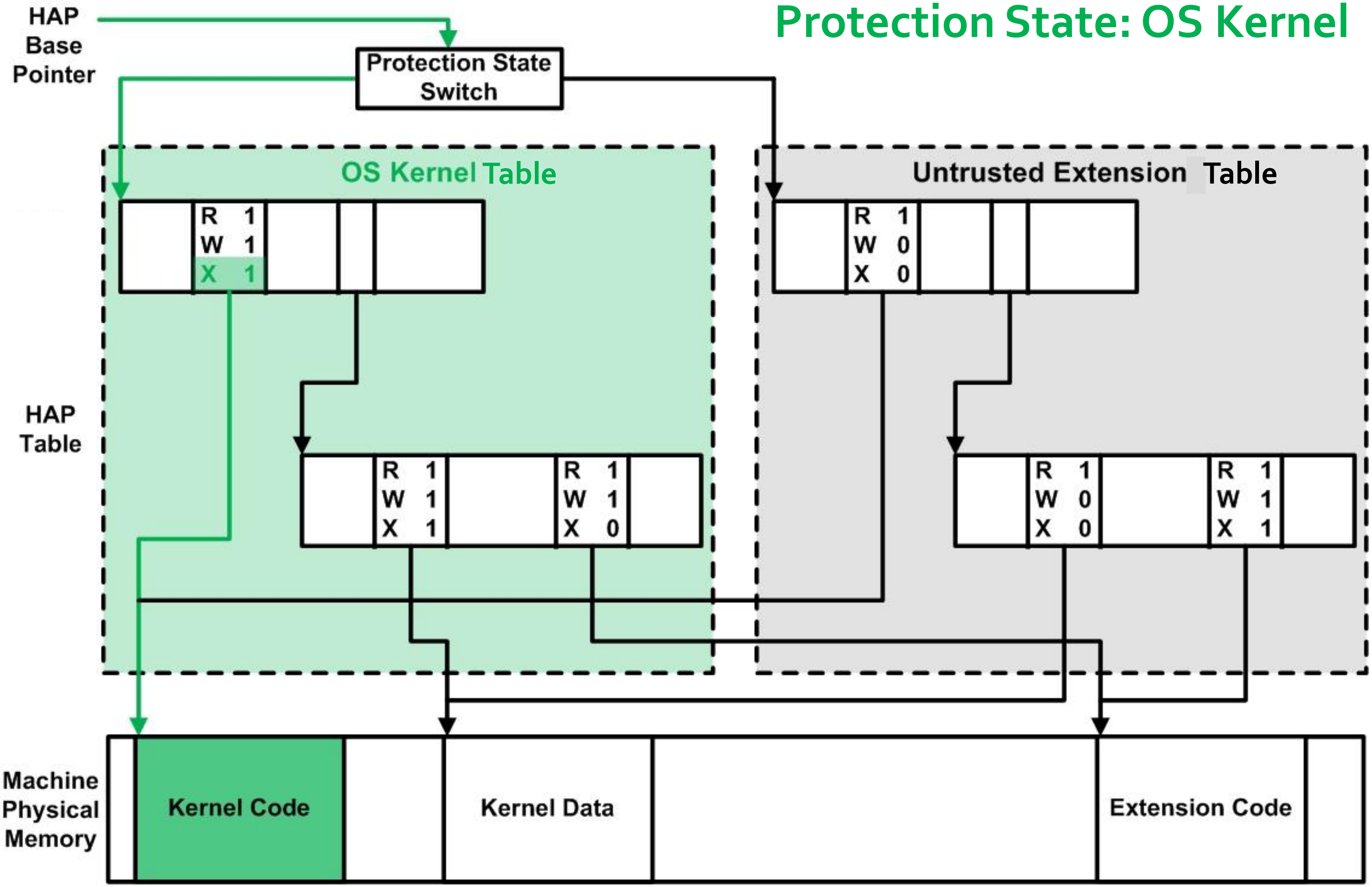
  - protection access rights are reflected in the page table access permissions

  - protection state transitions can be caught by setting **execution** permissions

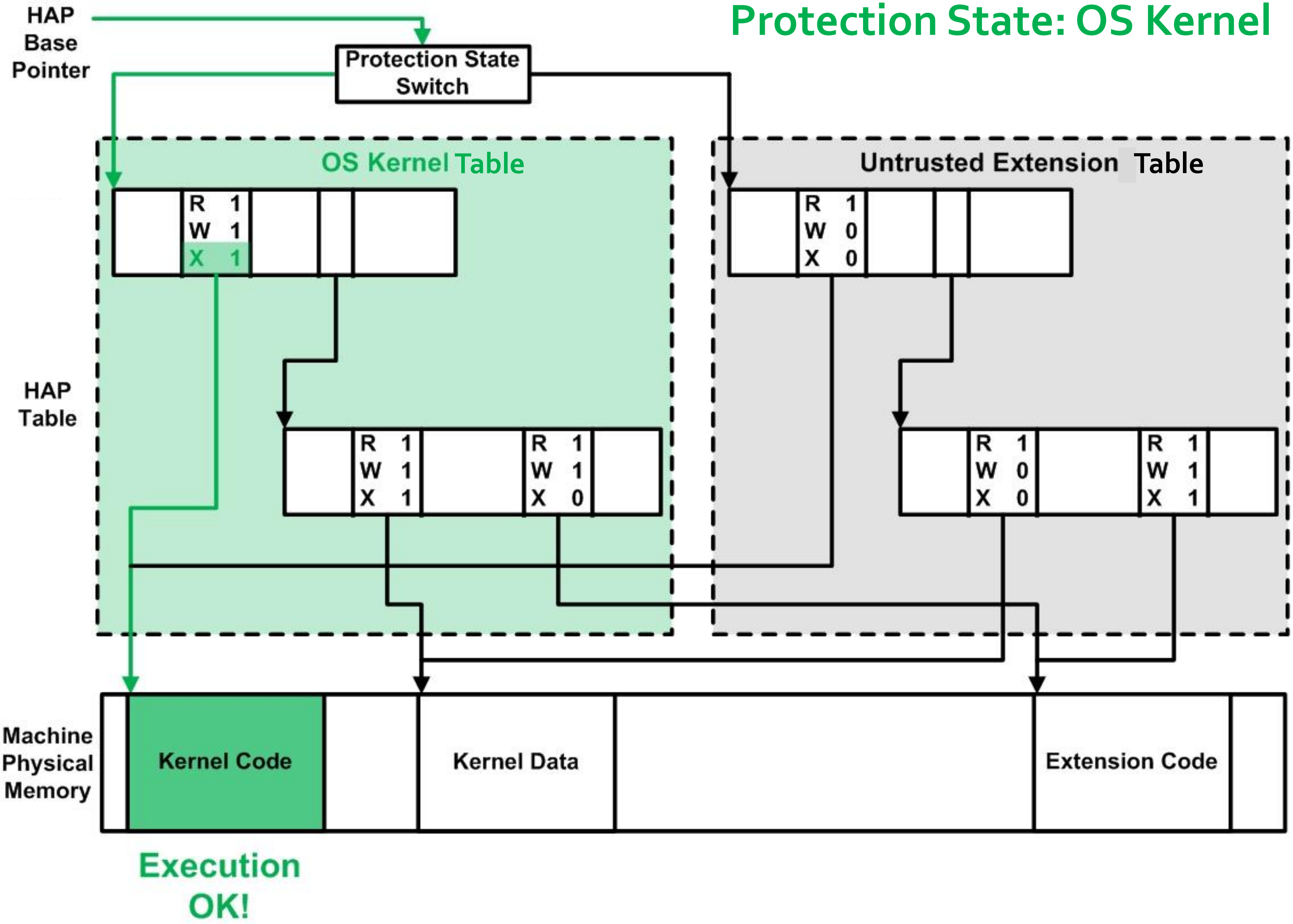
Example work flow

- Components
  - Protection states
  - Object labeling
  - Memory isolation

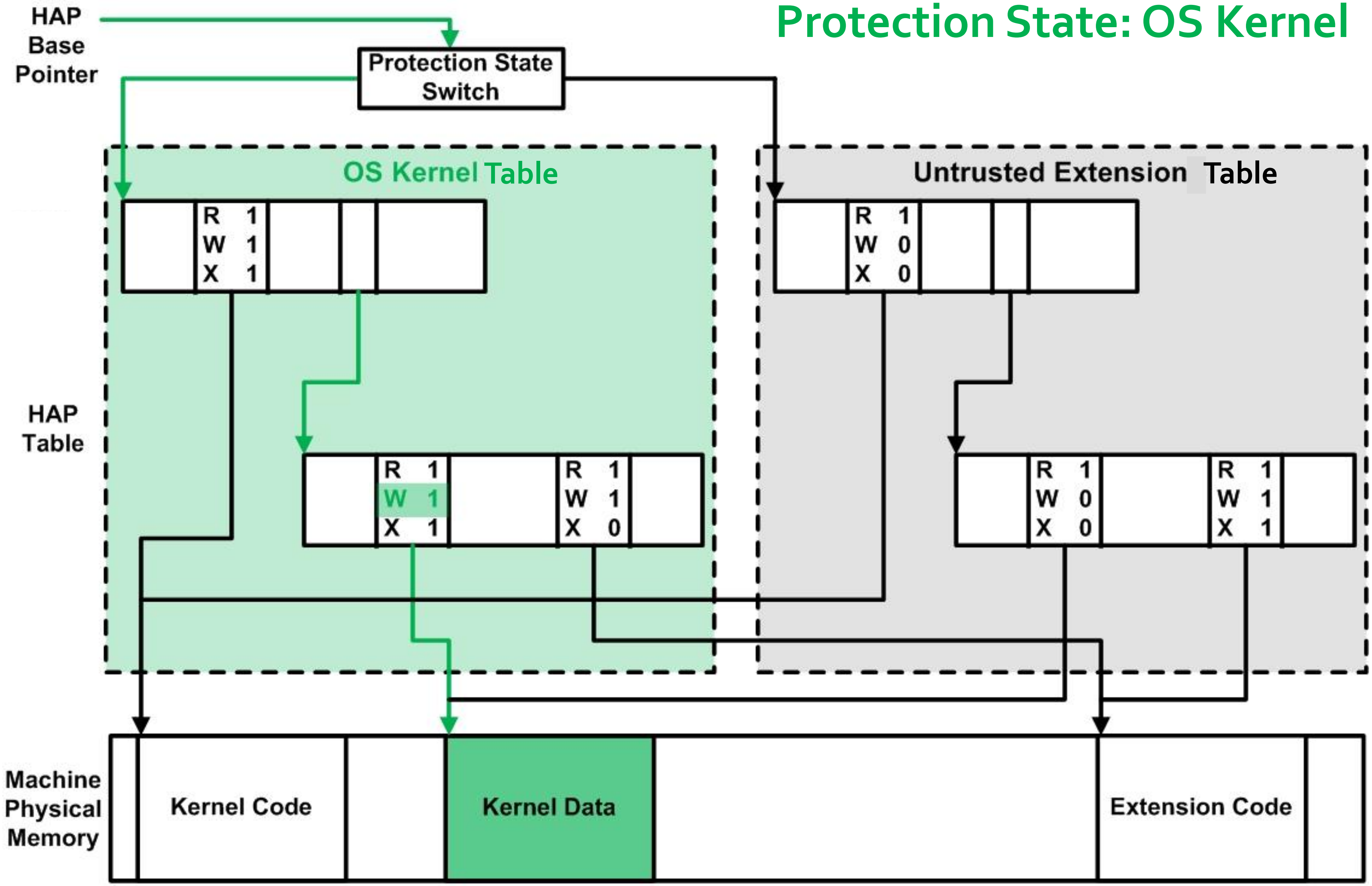
Protection State: OS Kernel



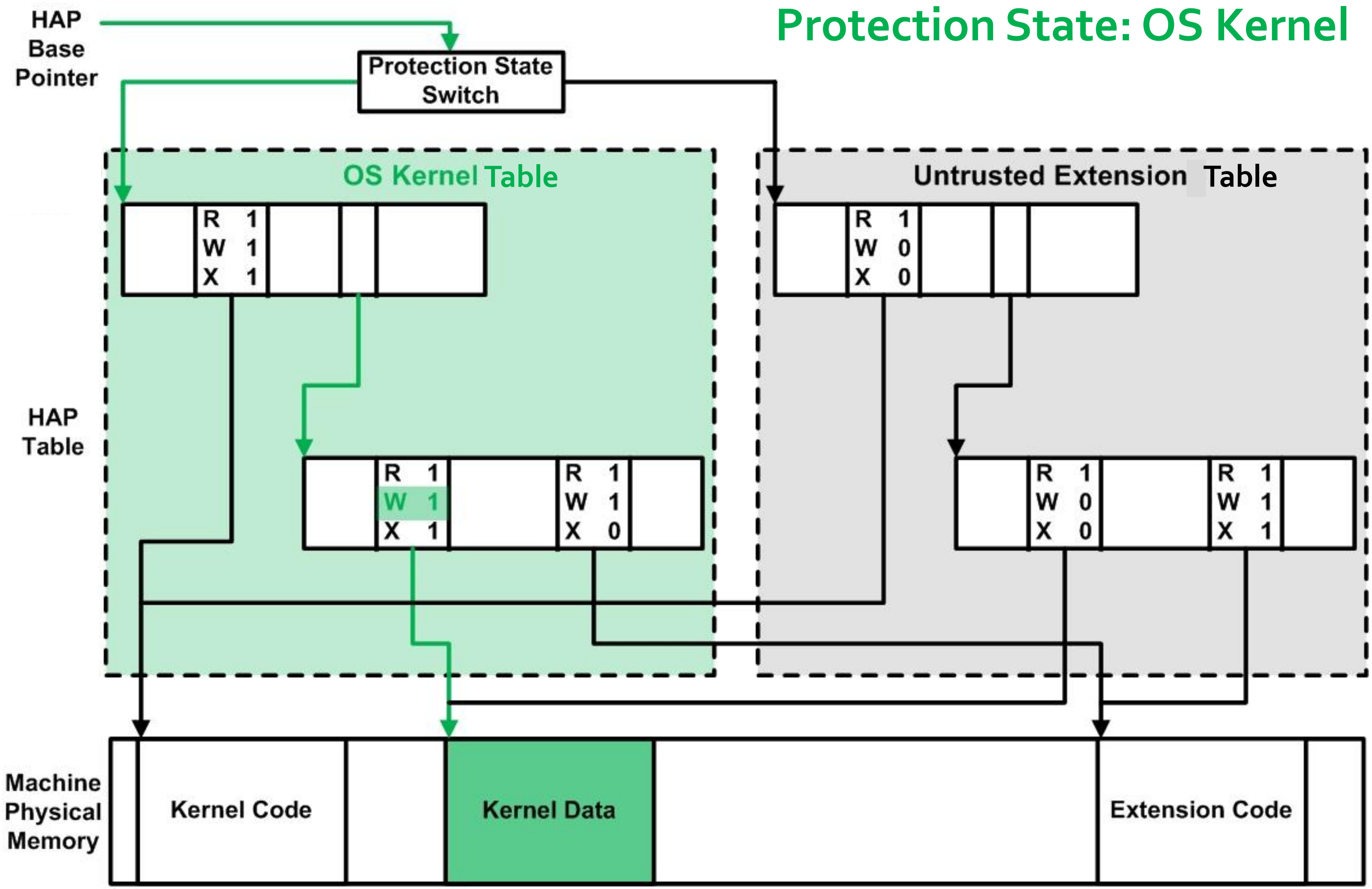
Protection State: OS Kernel



Protection State: OS Kernel

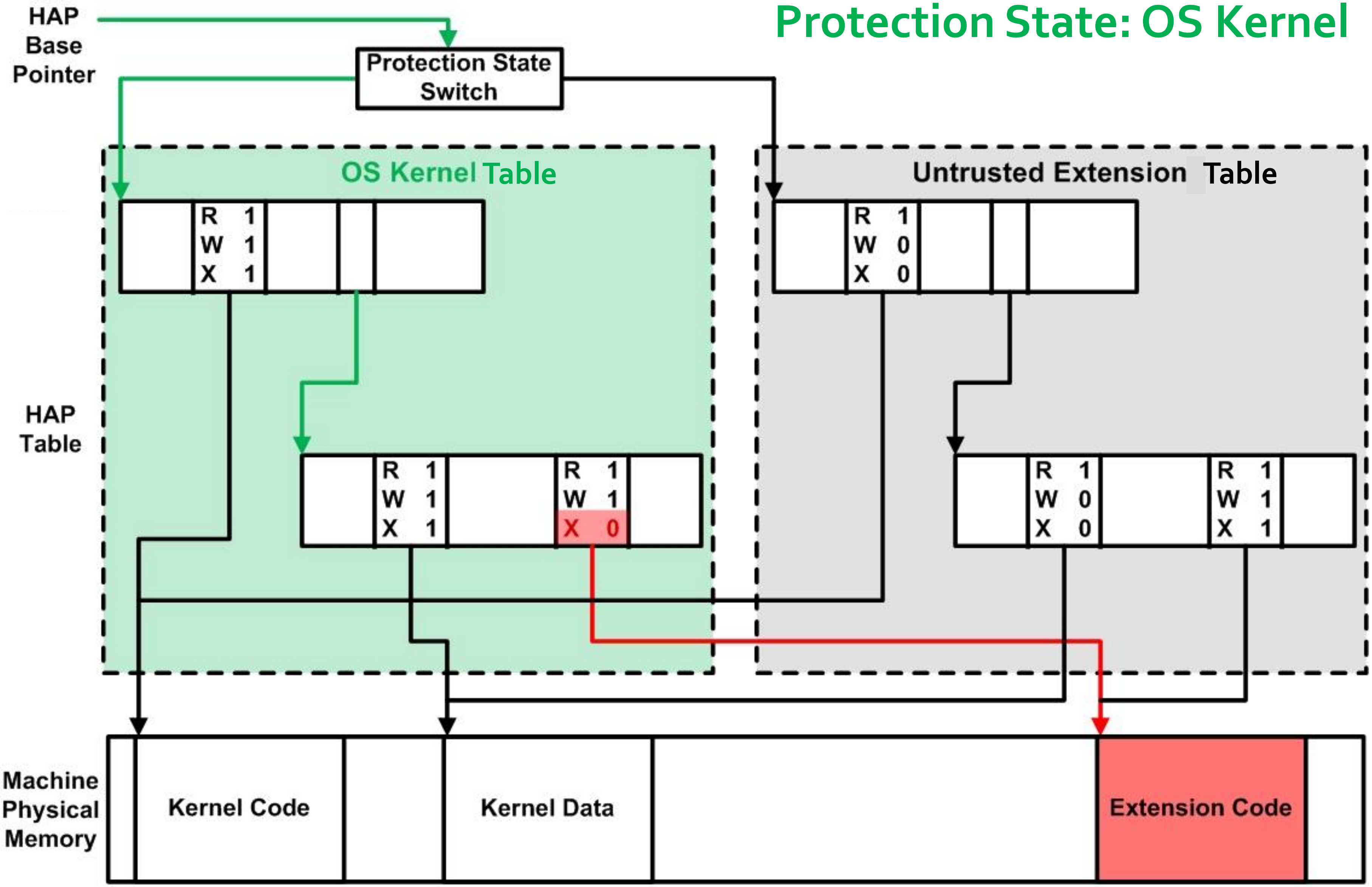


Protection State: OS Kernel

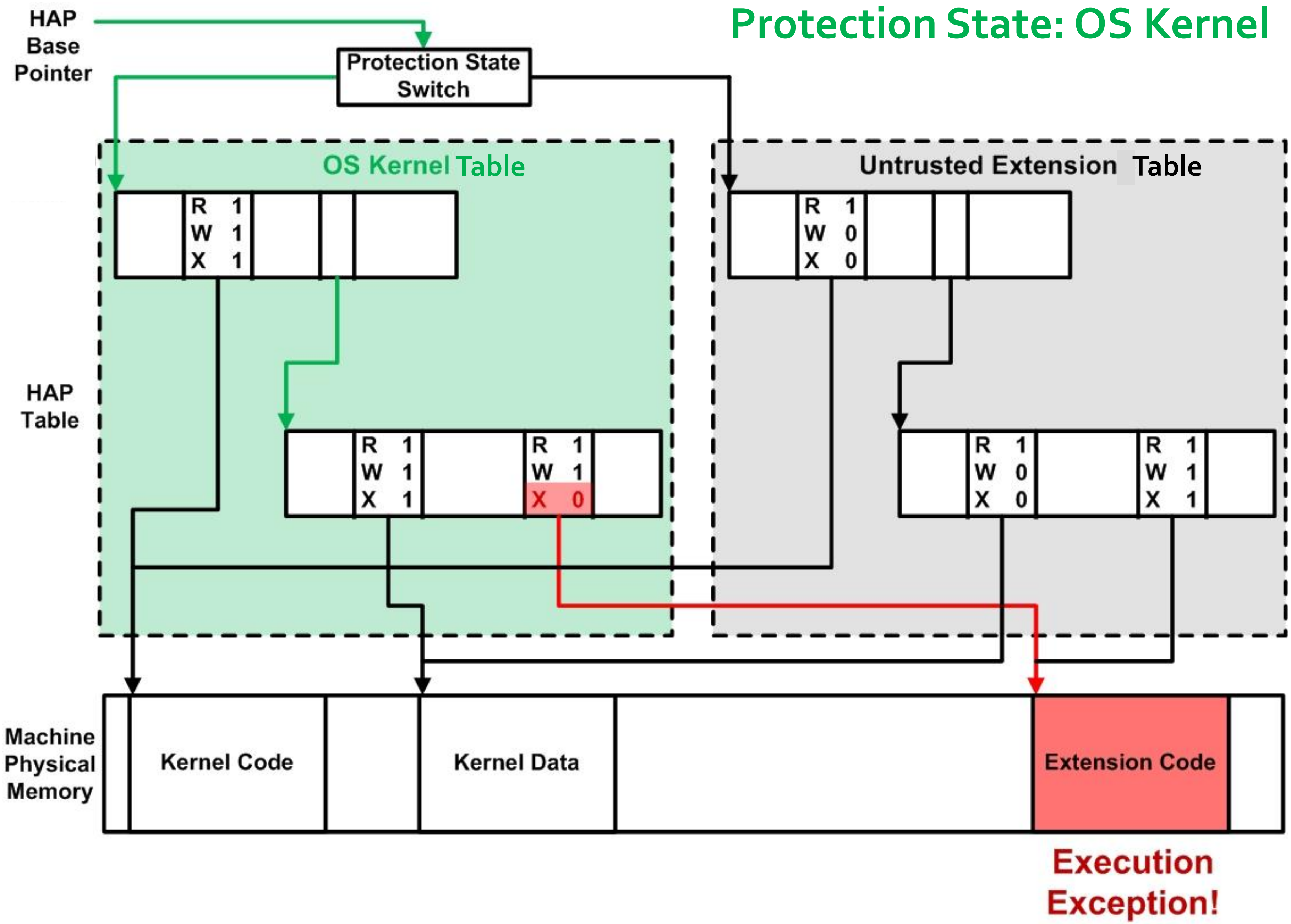


Write OK!

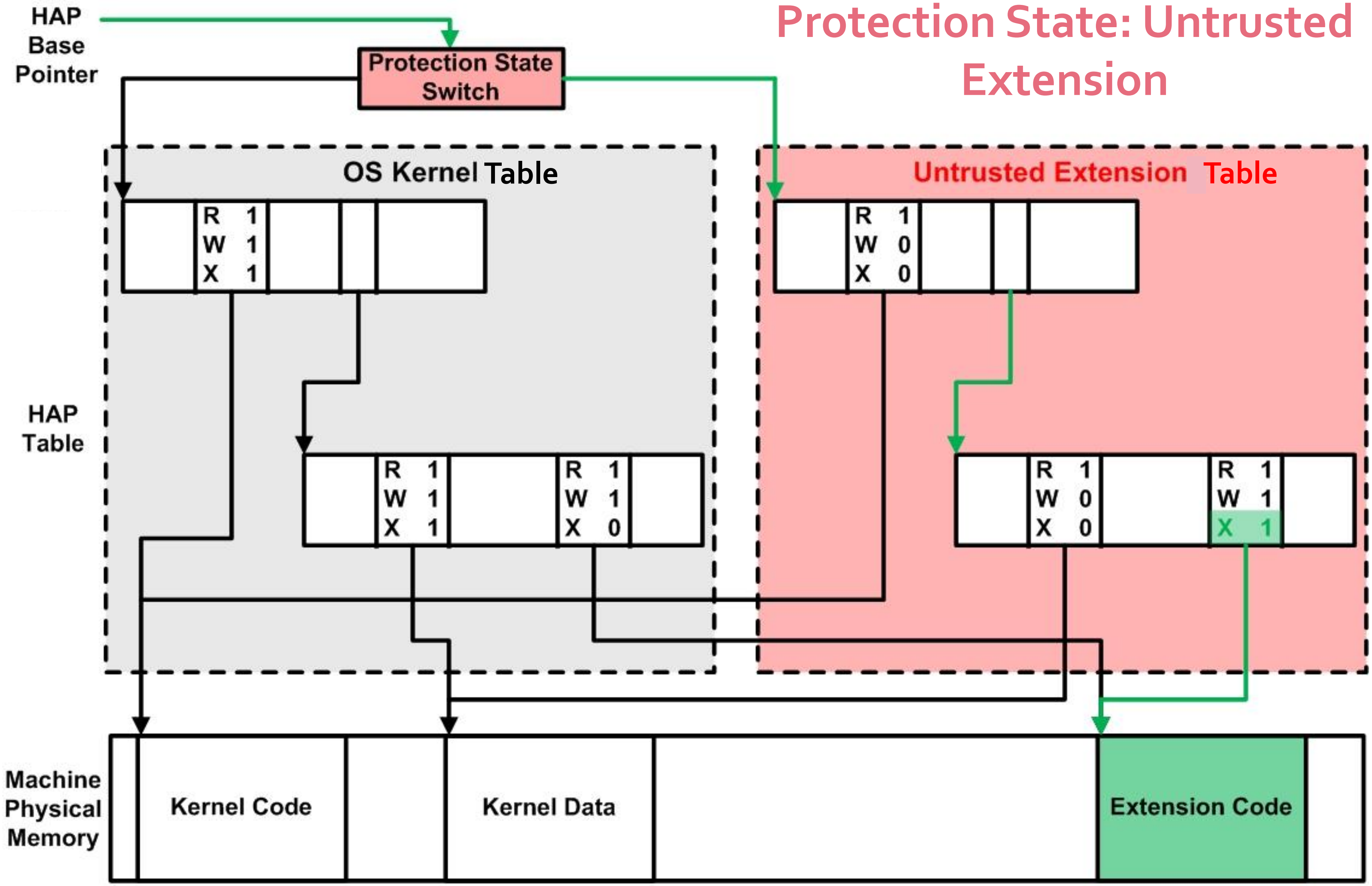
Protection State: OS Kernel



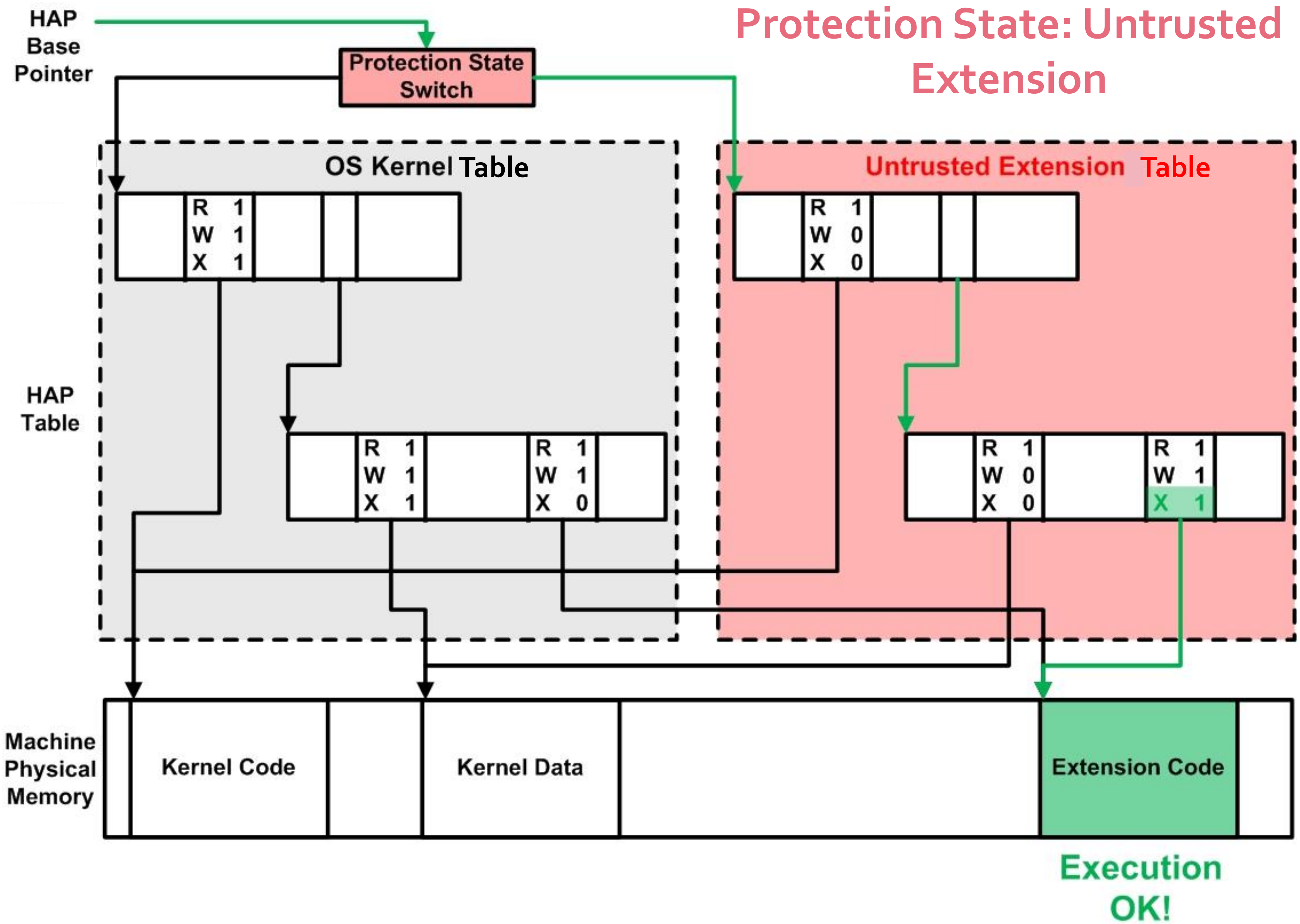
Protection State: OS Kernel



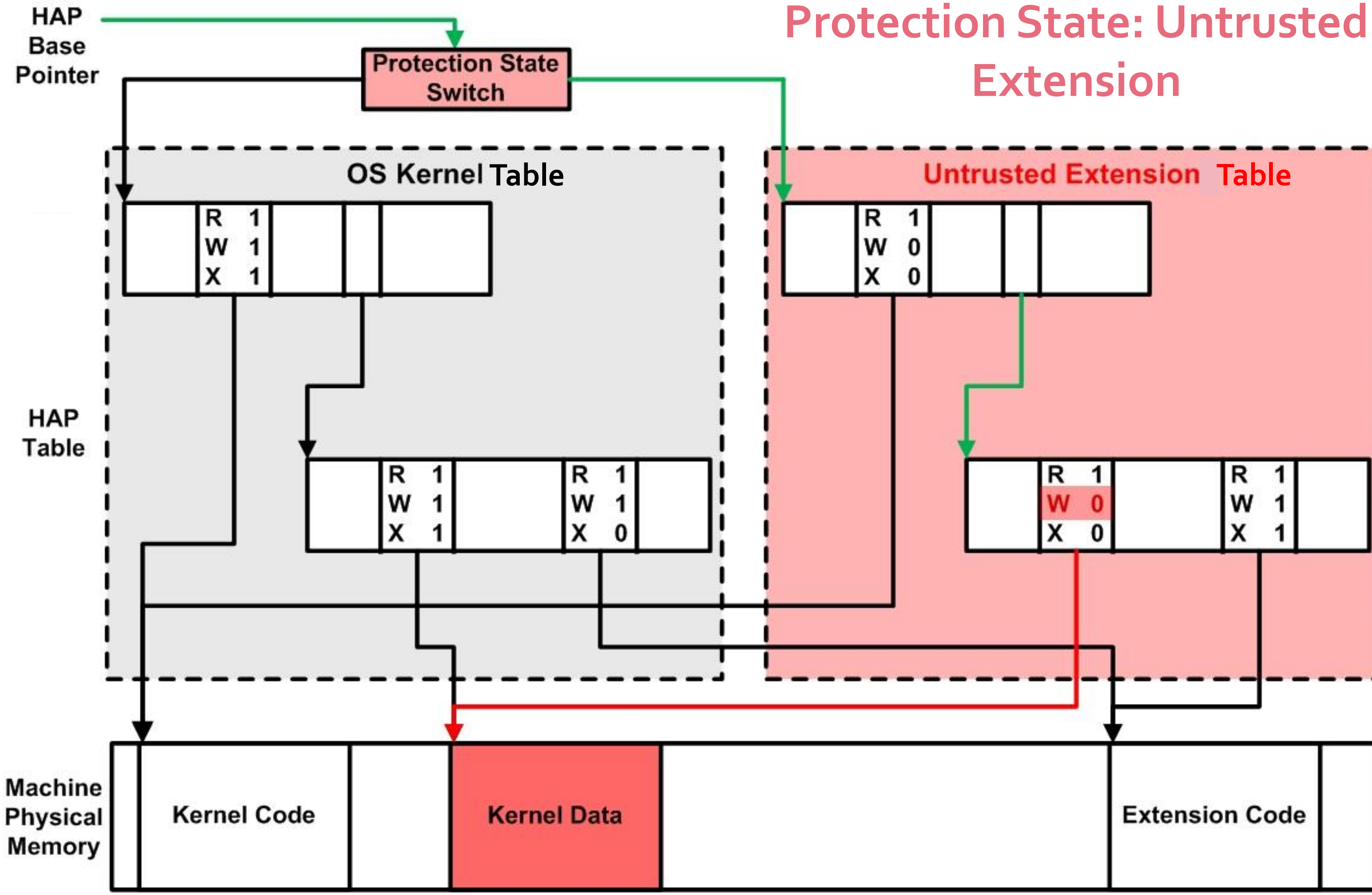
Protection State: Untrusted Extension



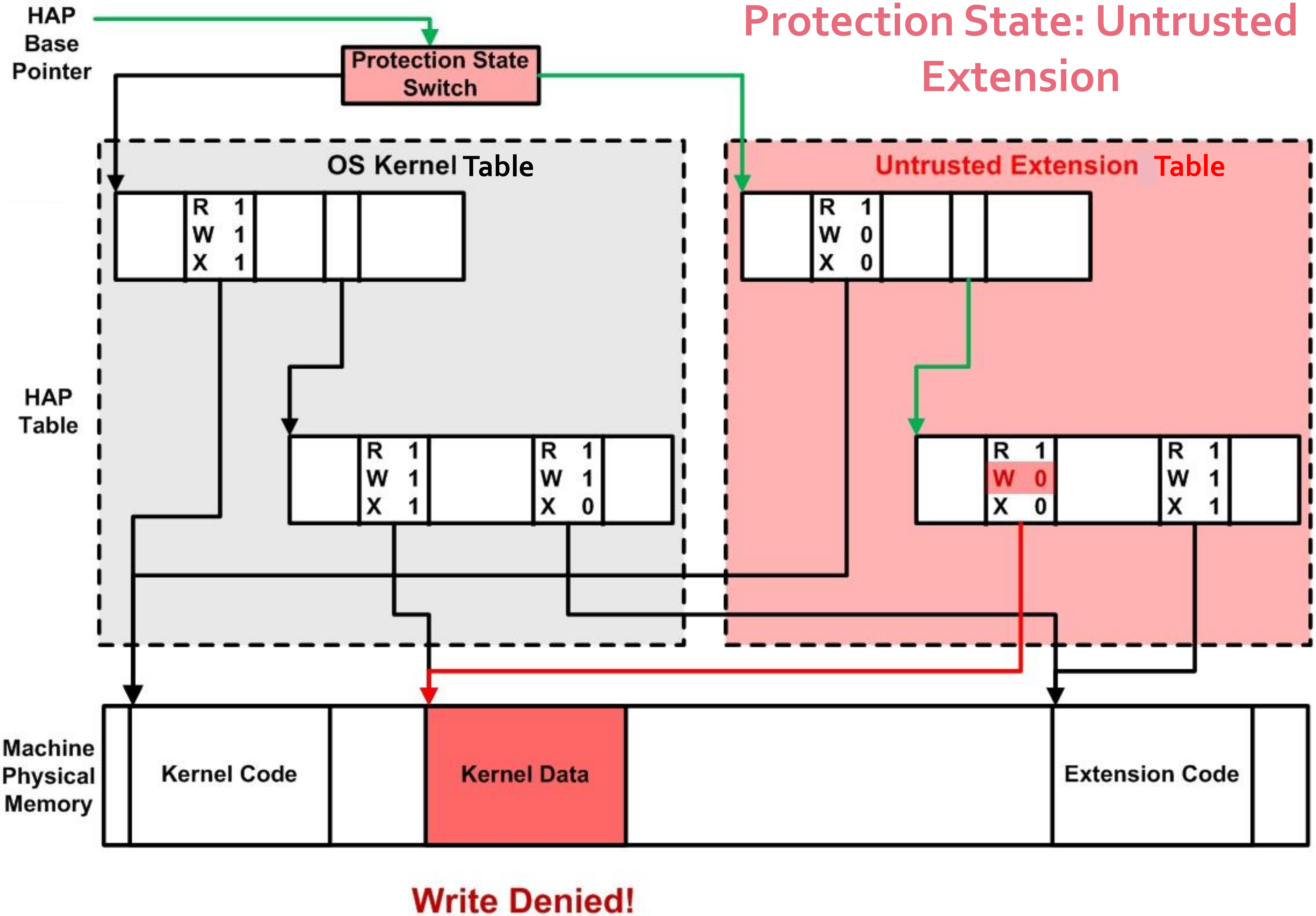
Protection State: Untrusted Extension



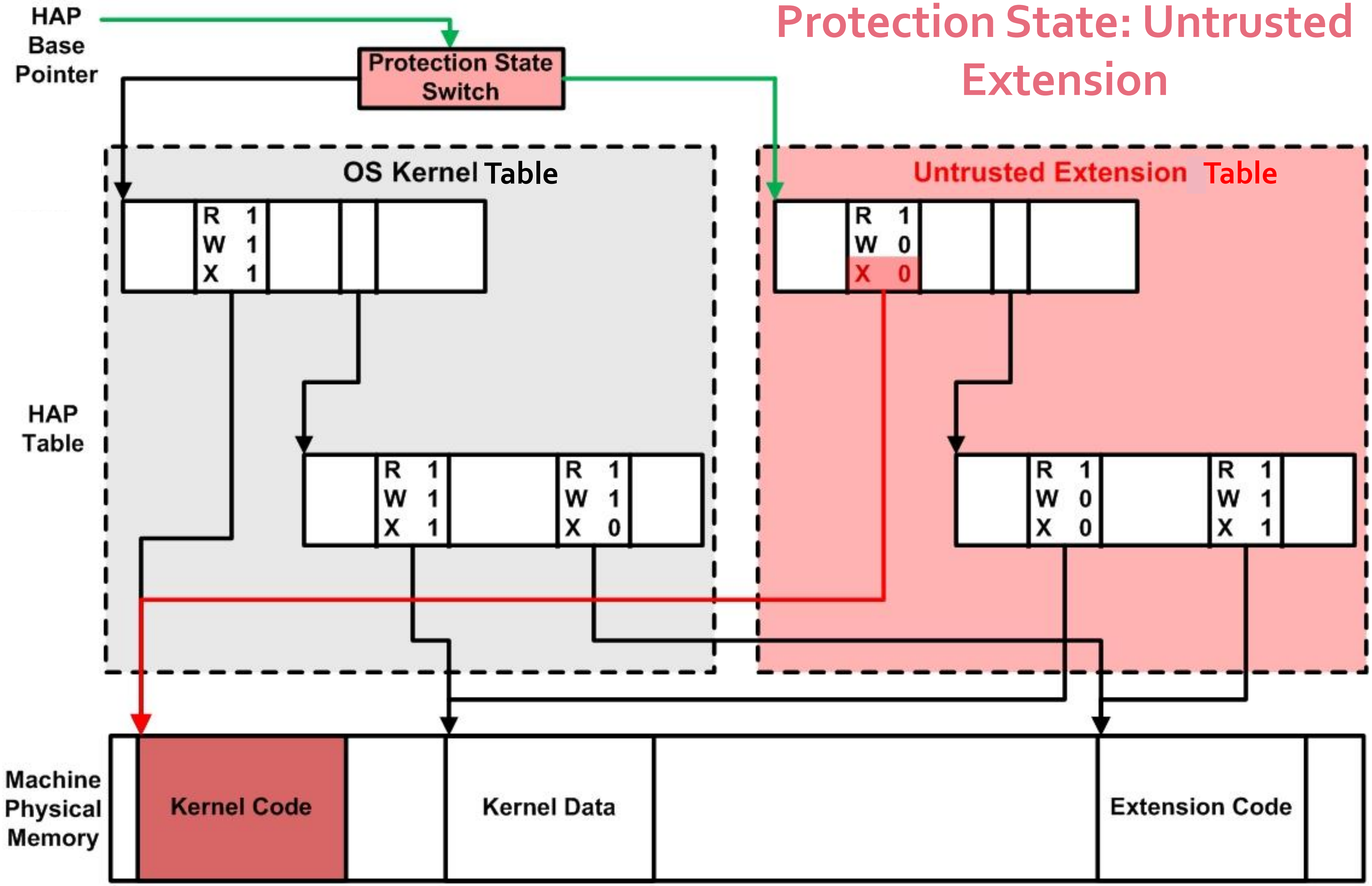
Protection State: Untrusted Extension



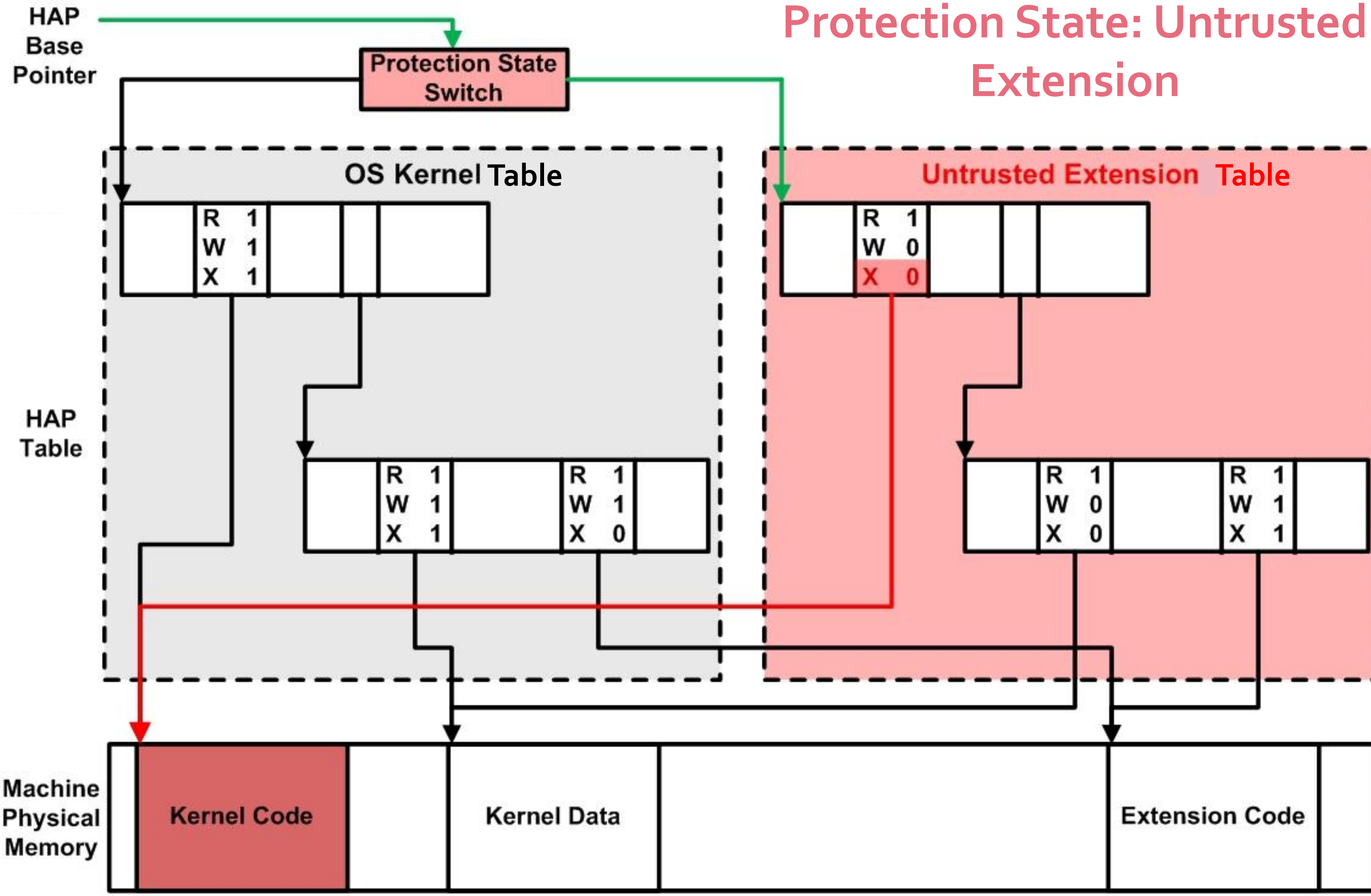
Protection State: Untrusted Extension



Protection State: Untrusted Extension

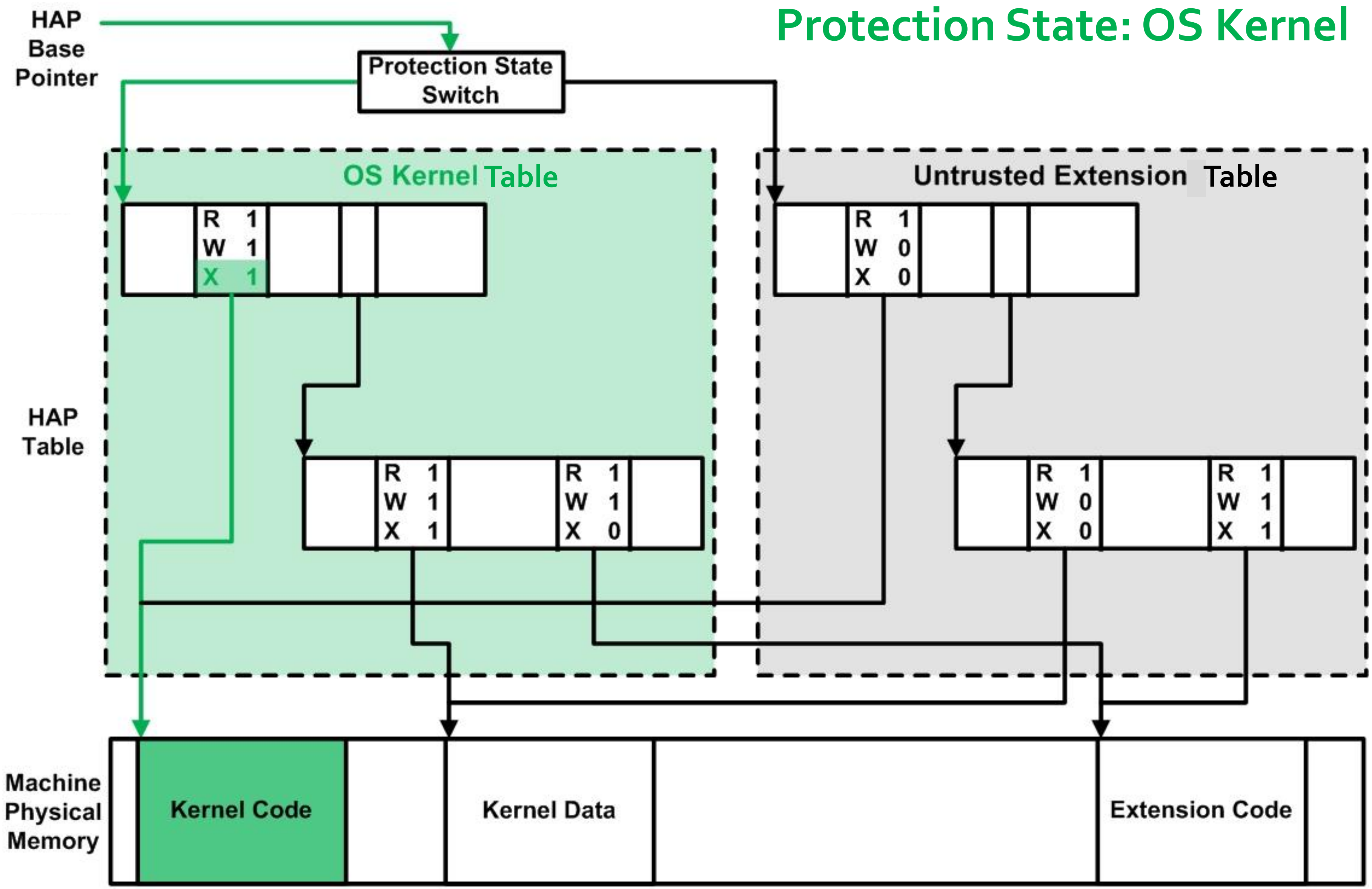


Protection State: Untrusted Extension



Execution Exception!

Protection State: OS Kernel



Implementation

- Prototype built on Intel's Extended Page Table (EPT) and Xen hypervisor 3.4.2

- Utilize unused bits in EPT entry for page label

- a trusted Linux kernel module to gather information from dynamic allocators and module loader
  - facilitate object labeling

HAP vs. Shadow Paging

- In our opinion, HAP is a cleaner design solution
  - Independent layer, do not need to be consistent with guest page tables
  - Less update, easier to synchronize multiple copies
  - Less unnecessary VMEXITs
    - Do not need to trap guest CR3 and GPT modifications
  - Better TLB performance

Other Issues

- Stack Integrity
  - create private stack frames by leveraging Multi-HAP
  - only writes in its own frames are propagated to the real kernel stack
- Write through DMA
  - IOMMU (Intel VT-d) page tables
- Architectural state integrity
  - save architectural state to VMM before transition to untrusted extension

Control Flow Integrity

- Access control for control flow transfers between untrusted extensions and OS kernel
  - All protection state transitions are intercepted by the hypervisor.
  - kernel control data (e.g., function pointer) are protected by the isolation mechanism
  - Kernel stack frames are also guarded.

Control Flow Integrity

- Trusted Entry Points are a set of addresses specified by OS developer or administrator
  - e.g., a tailored set of kernel APIs to confine certain category of extensions
- Other issues
  - Extension returns to kernel
    - maintain call-return consistencies
  - Interrupt and preemption

Outline

- Motivation
- Approach Overview
- Key Design & Implementation
- **Evaluation**
- Summary

Evaluation - Security

- Security Analysis
  - Change kernel code
    - detected by code integrity protection
  - Modify kernel control / non-control data
    - detected by data integrity protection
  - Manipulate return addresses / kernel stack frames
    - call-return inconsistencies
    - Kernel stack frame protection

- Evaluated with both real-world and homegrown malicious extensions

Evaluation - Performance

Benchmark	Untrusted Extensions	# of Protection State Transfers	Native Performance	HUKO Performance	Relative Performance
Dhrystone 2	8139too, ext3	N/A	10, 855, 484 lps	10, 176, 782 lps	0.94
Whetstone	8139too, ext3	N/A	2, 270 MWIPS	2, 265 MWIPS	1.00
Lmbench (pipe bandwidth)	8139too, ext3	N/A	2, 535 MB/s	2, 213 MB/s	0.87
Apache Bench	8139too	56, 037	2, 261 KB/s	1, 955 KB/s	0.86
Kernel Decompression	ext3	17, 471, 989	35, 271 ms	44, 803 ms	0.79
Kernel Build	ext3	148, 823, 045	2, 804 s	3, 106 s	0.90

Evaluation - Performance

- Major performance cost: protection state transitions
  - Involves guest-to-VMM switch (VMEXIT)

- The more frequent untrusted extension interacts with the kernel, the larger performance penalties

Outline

- Motivation
- Approach Overview
- Key Design & Implementation
- Evaluation
- **Summary**

Reliability for buggy device drivers

- Microkernels
  - L4 [Liedtke `95], MINIX 3 [Herder `09]
- Device driver isolation
  - Nooks [Swift '03], Mondrix [Witchel `05]
- Software fault isolation
  - XFI [Erlingsson `06]

Limitation & Future Work

- Labeling Objects at the page-level
  - trade-off: performance vs. security
- Kernel API not designed for isolation/sandboxing
  - invoking APIs may violate integrity properties
  - may need sanitizing & privilege separation
- Tune the OS Kernel
  - e.g., eliminates mixed pages to improve security and efficiency

Thanks!

Questions?

Summary

- HUKO significantly limits the attacker's ability to compromise the integrity of the kernel.
- Contemporary hardware features may facilitate sandboxing and reference monitoring in the kernel space.