Introduction to Platform Security

BJÖRN RUYTENBERG
EINDHOVEN UNIVERSITY OF TECHNOLOGY

Who Am I

Björn Ruytenberg
@0Xiphorus

Security researcher

Main interests: hardware and firmware security, sandboxing, input validation

More about me: https://bjornweb.nl

MSc student in Computer Science @ TUE

Master thesis:
"Thunderspy – When Lightning Strikes Thrice: Breaking Thunderbolt 3 Security"

- Presented at Black Hat USA, Chaos Communication Congress and other venues
- More details in next lecture

TU/e

Roadmap

Introduction to Platform Security

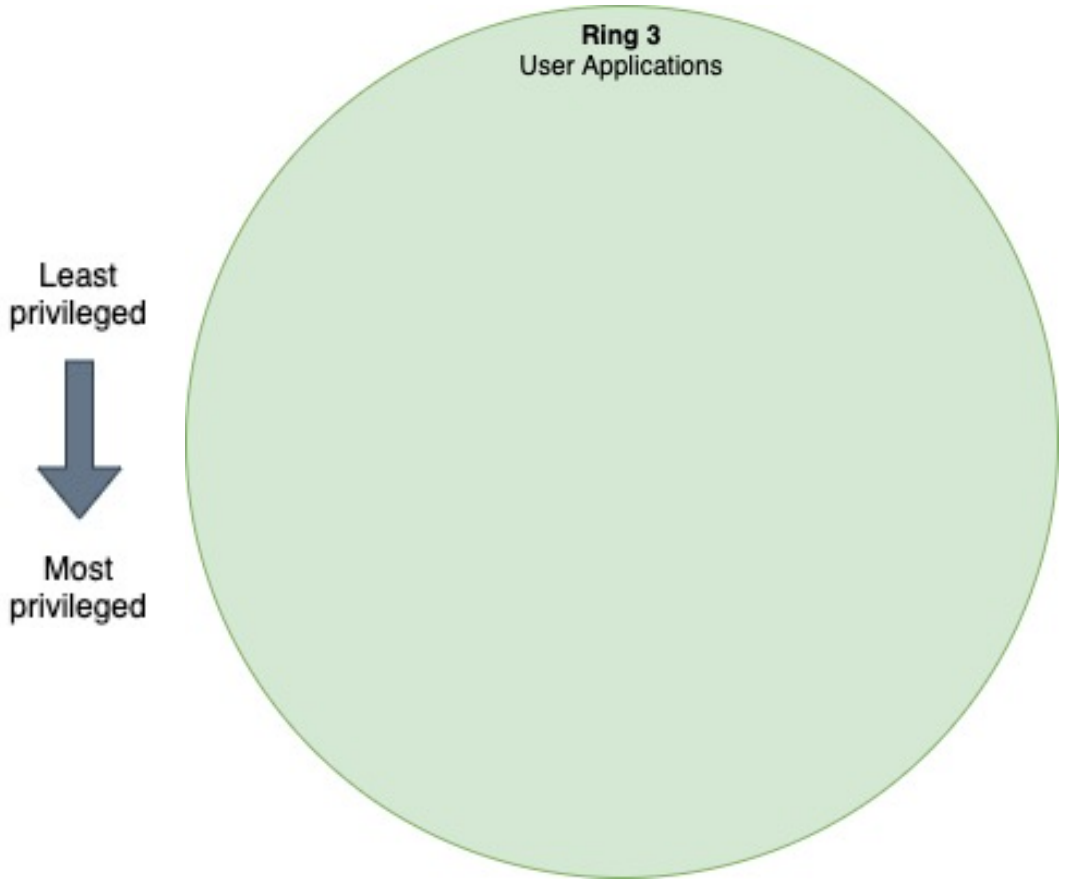
- Protection Rings
- x86 System Architecture
  - PCI Express Basics
  - PCI Express Security
  - System firmware (BIOS, UEFI)
  - Operating System kernel
  - Device drivers
- x86 Boot Process and Security
  - BIOS vs. UEFI boot
  - Secure Boot, Verified Boot, Measured Boot

Defense-in-Depth

- **Goal:** Protect information systems against adversaries, by
  - Identifying vulnerabilities pre-deployment
  - Mitigating vulnerabilities post-deployment through countermeasures
- Countermeasures are typically not perfect
  - May be circumvented
  - May introduce vulnerabilities of their own
- **Defense-in-Depth:** Ensure compromising one security control does not result in immediate, full system compromise; allow for fallback on additional controls

Protection Rings

Protection Rings

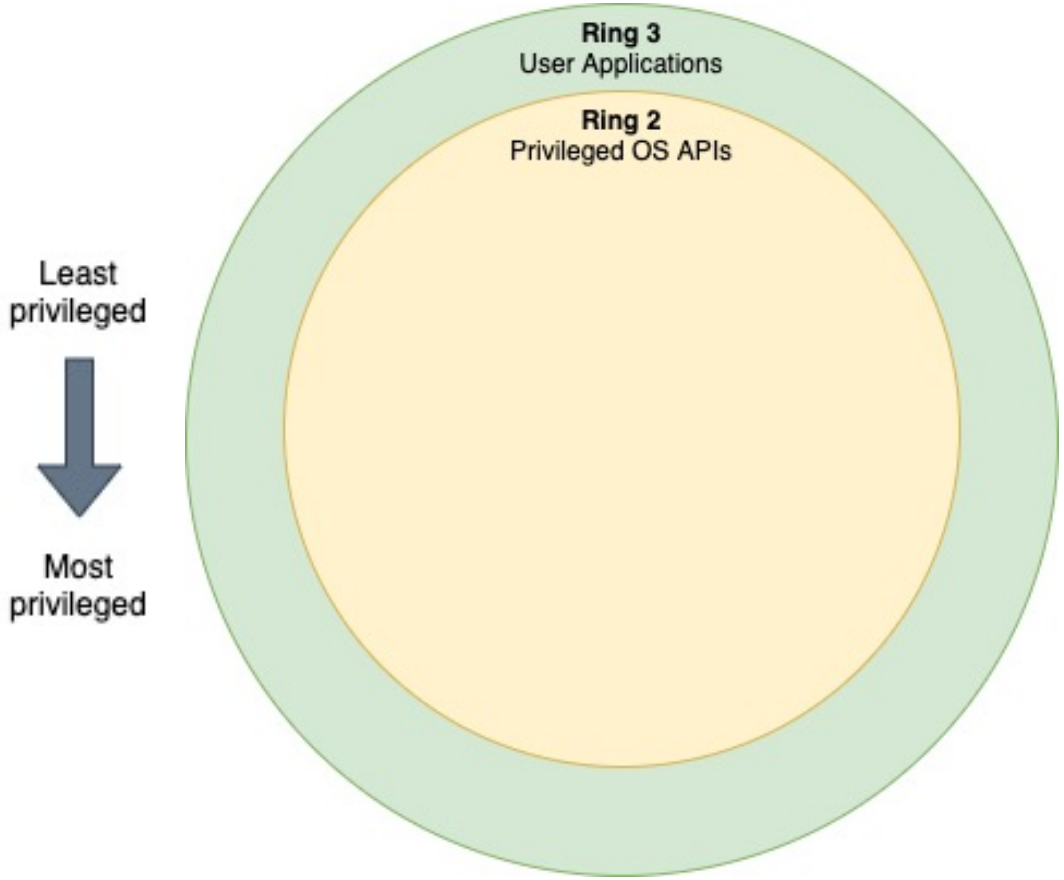


User Applications

- **Compile time mitigations:** strong typing, managed memory allocation, code review, static analysis, formal program verification, dynamic analysis (e.g. ASAN, UBSAN), unit testing
- **Runtime mitigations:** ASLR, NX, stack canaries, shadow stacks, CFI, language runtime sandboxing
- **OS-provided runtime mitigations:**
  - User vs. kernel space: virtualize memory, IPC and hardware I/O
  - Limit process privileges through
    - Access control (user, file system, group policies, capabilities)
    - Process sandboxing (jailing/chrooting, containerization)

TU/e

Protection Rings

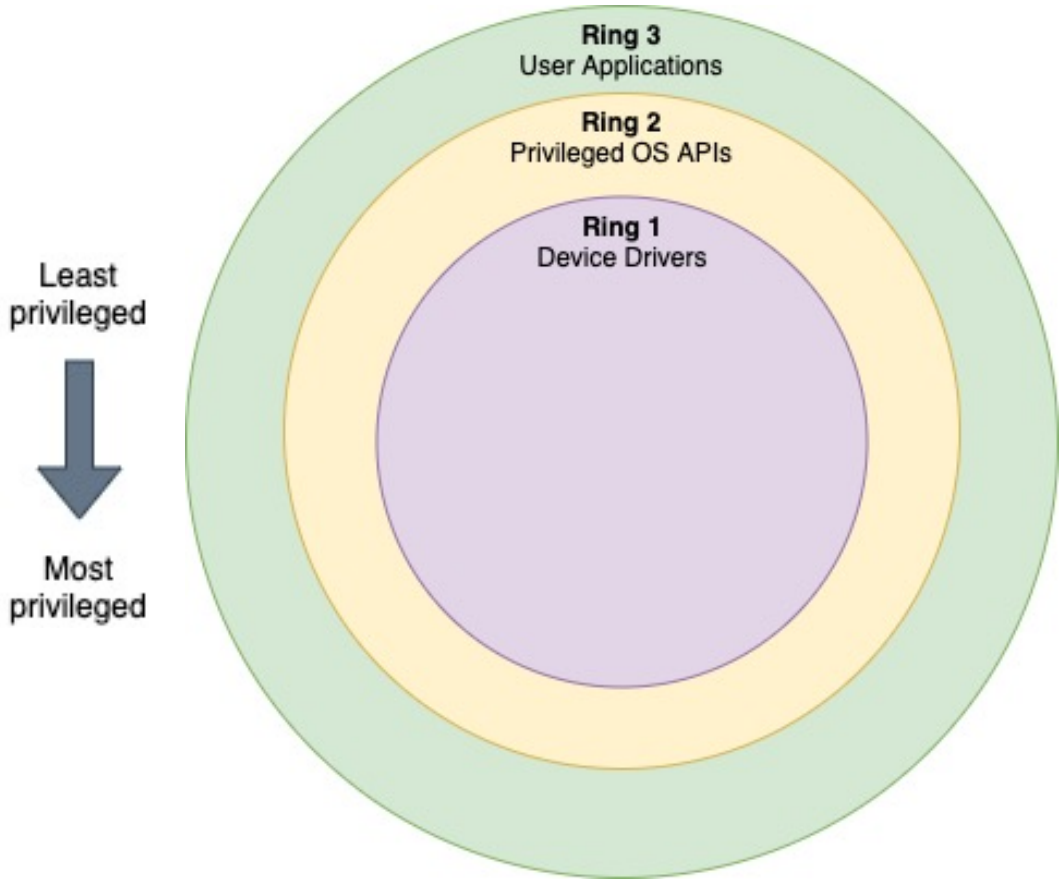


Operating System APIs

- User space cannot perform memory management, filesystem and driver I/O directly
- Consumes OS APIs, possibly through language runtime, such as
  - Memory I/O, filesystem I/O, IPC: sockets, syscalls
  - Graphics and sound subsystems: DirectX, OpenGL, Vulkan
  - Driver interfacing: WMI, IOCTLs, block+char devices
- May require elevated privileges (e.g. root, capability provisioning)

TU/e

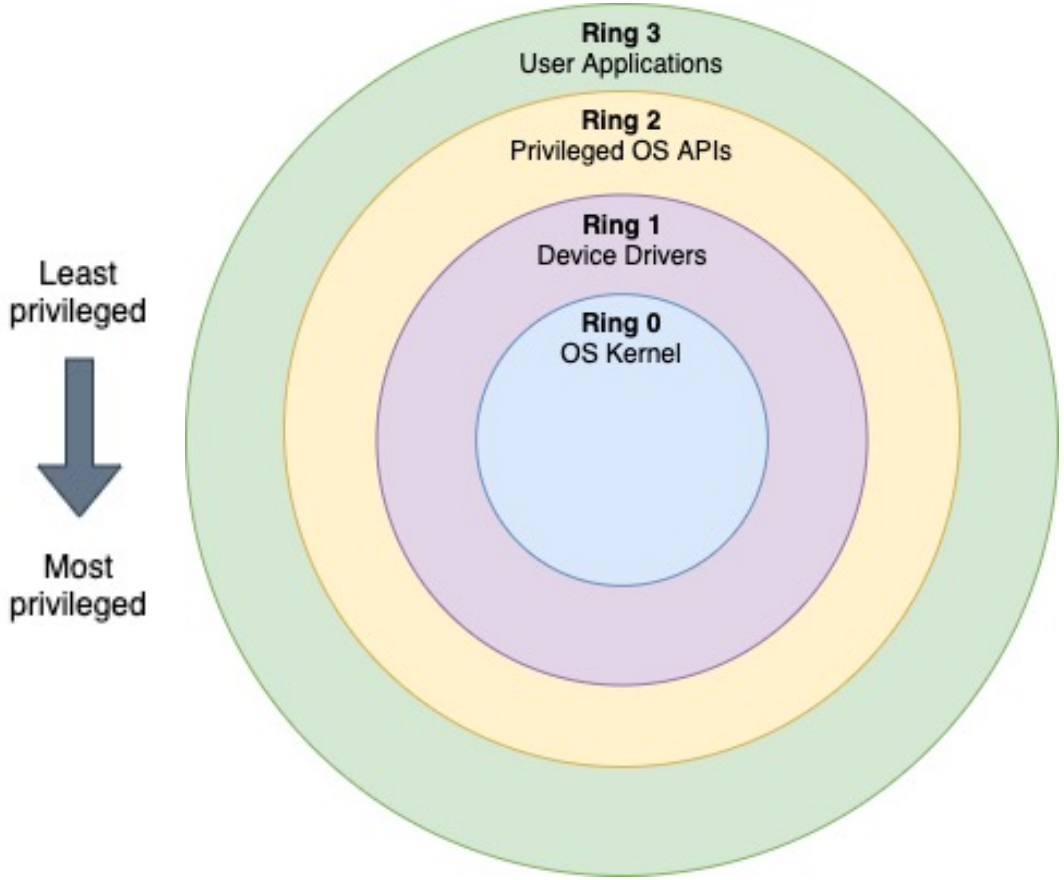
Protection Rings



Device drivers

- GPU, NIC, HD Audio, USB host controllers + devices, internal storage

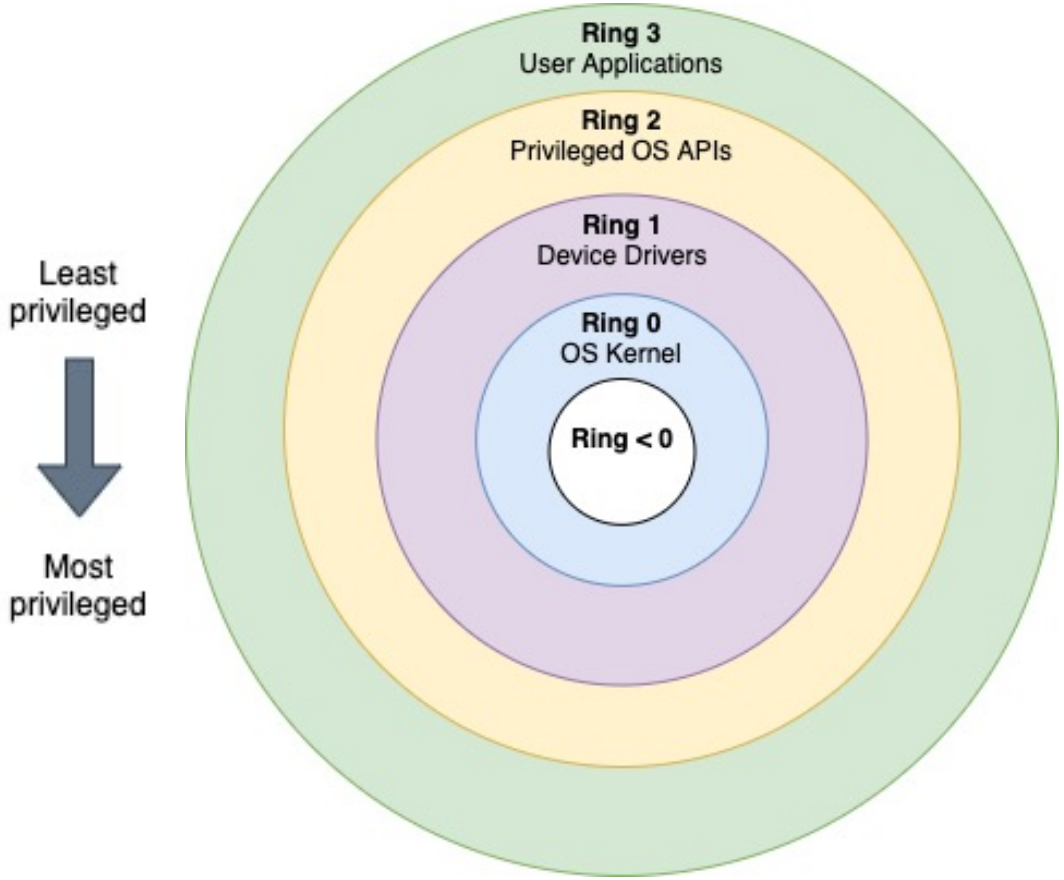
Protection Rings



OS Kernel

- Implements OS and device driver APIs
- Final user-controllable software layer

Protection Rings

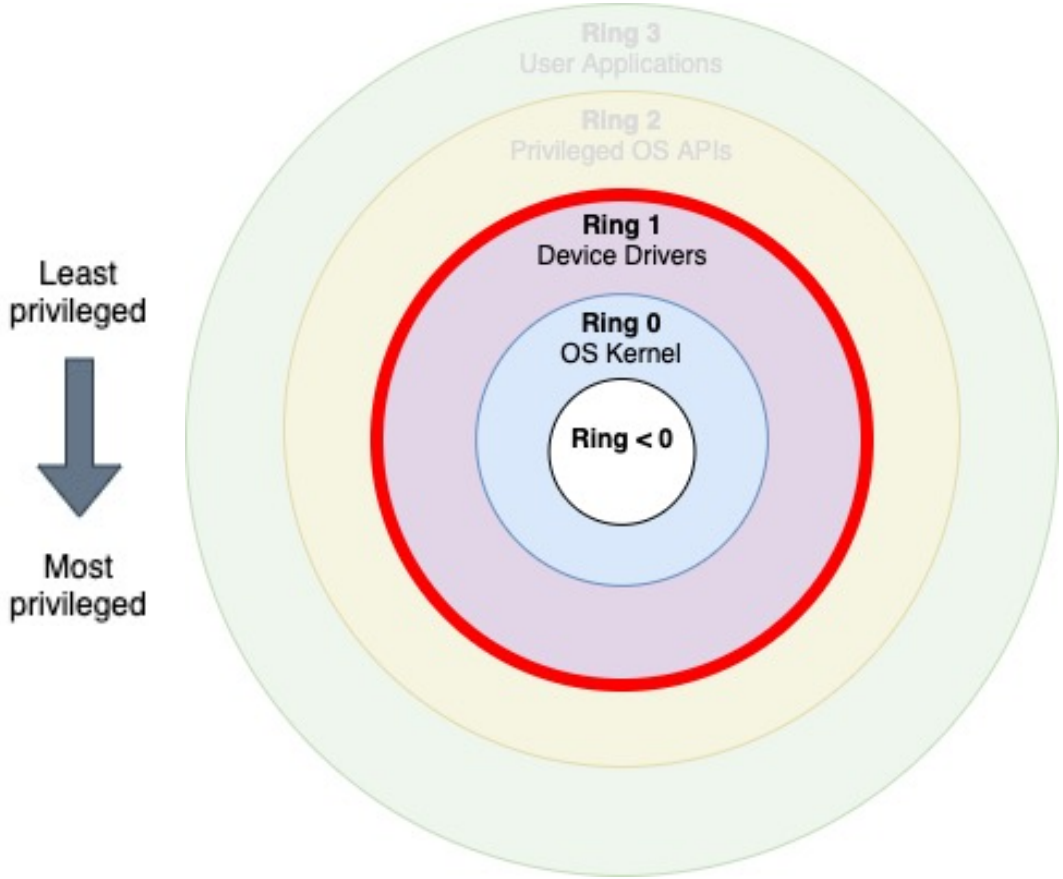


Ring < 0

- Hypervisors (virtualization)
- Firmware (BIOS, UEFI)
- System Management Mode (SMM)
- Co-processors, including
  - Cryptography + boot process attestation: TPMs (more later), Intel ME
  - Trusted Execution Environments: Intel SGX, AMD PSP
- Silicon: CPU, GPU, storage, …

TU/e

Protection Rings



Platform Security

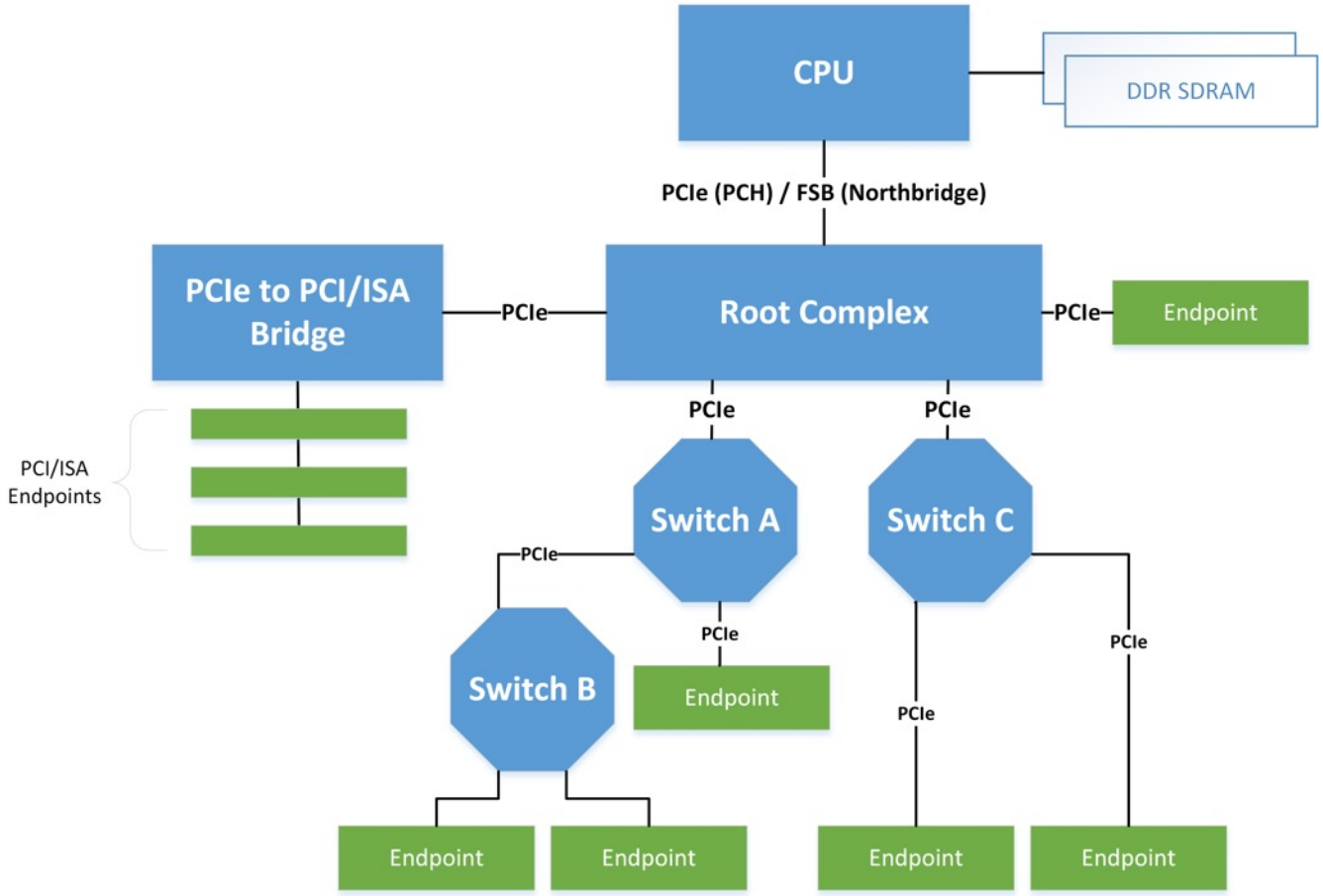
- Operates at the intersection of software and hardware security
- Focuses on Ring 1 and below

PCI Express Essentials

PCI Express Architecture

Network Topology

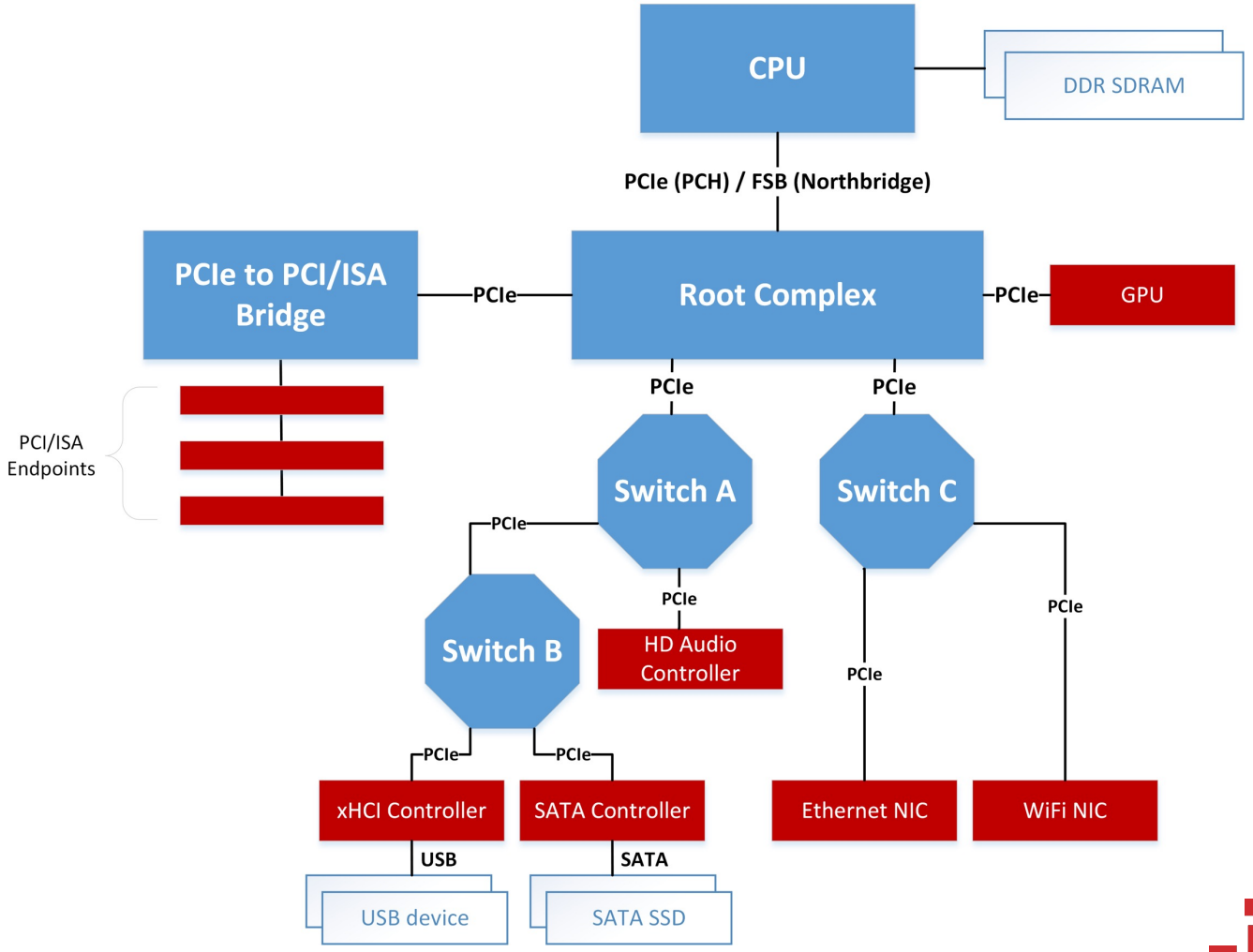
- Root Complex
- Switch
- Endpoints
- PCIe to legacy bridge (e.g. ISA/PCI/PCI-X)



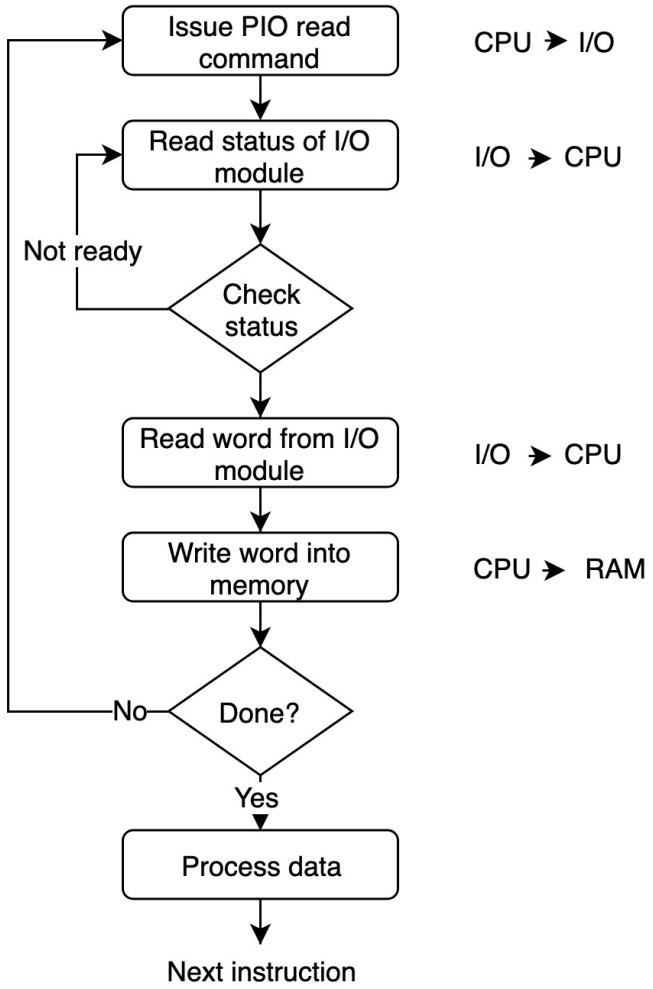
PCI Express Architecture

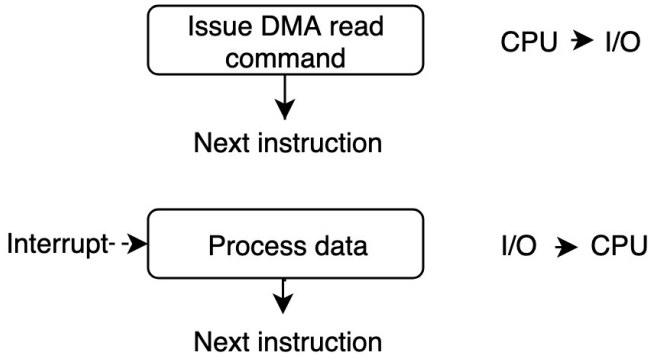
Endpoints

- GPU
- HD Audio Controller
- {O,E,X}HCI Controller (USB)
- SATA Controller
- Ethernet/WiFi NIC
- …



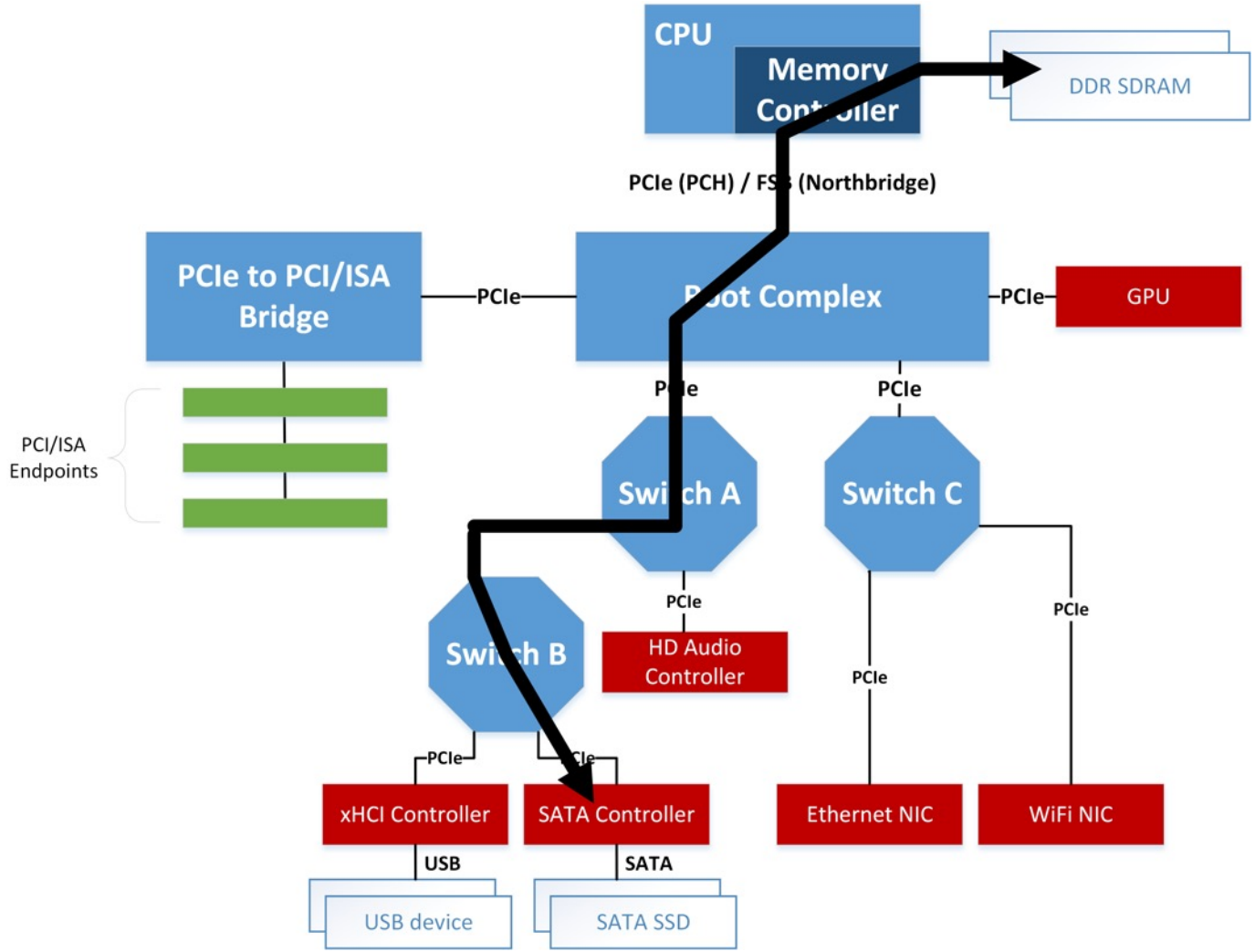
Pit stop: Programmed I/O vs. Direct Memory Access





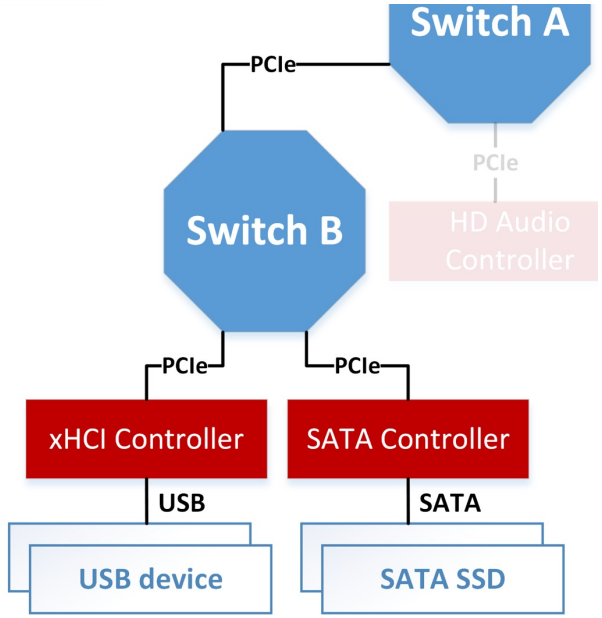
[Based on Hennessey et al. – Computer Architecture, A Quantitative Approach]

Pit stop: Programmed I/O vs. Direct Memory Access



Evolution of PCI Express: Legacy Stack

- Purpose-tailored peripheral interfaces separate devices from PCIe network
- Primary incentives:
  - Reduces redundant R&D on controller hardware
  - Interface support guarantees device compatibility with system
  - Interface support ensures (basic) device functionality is available without requiring vendor-specific drivers

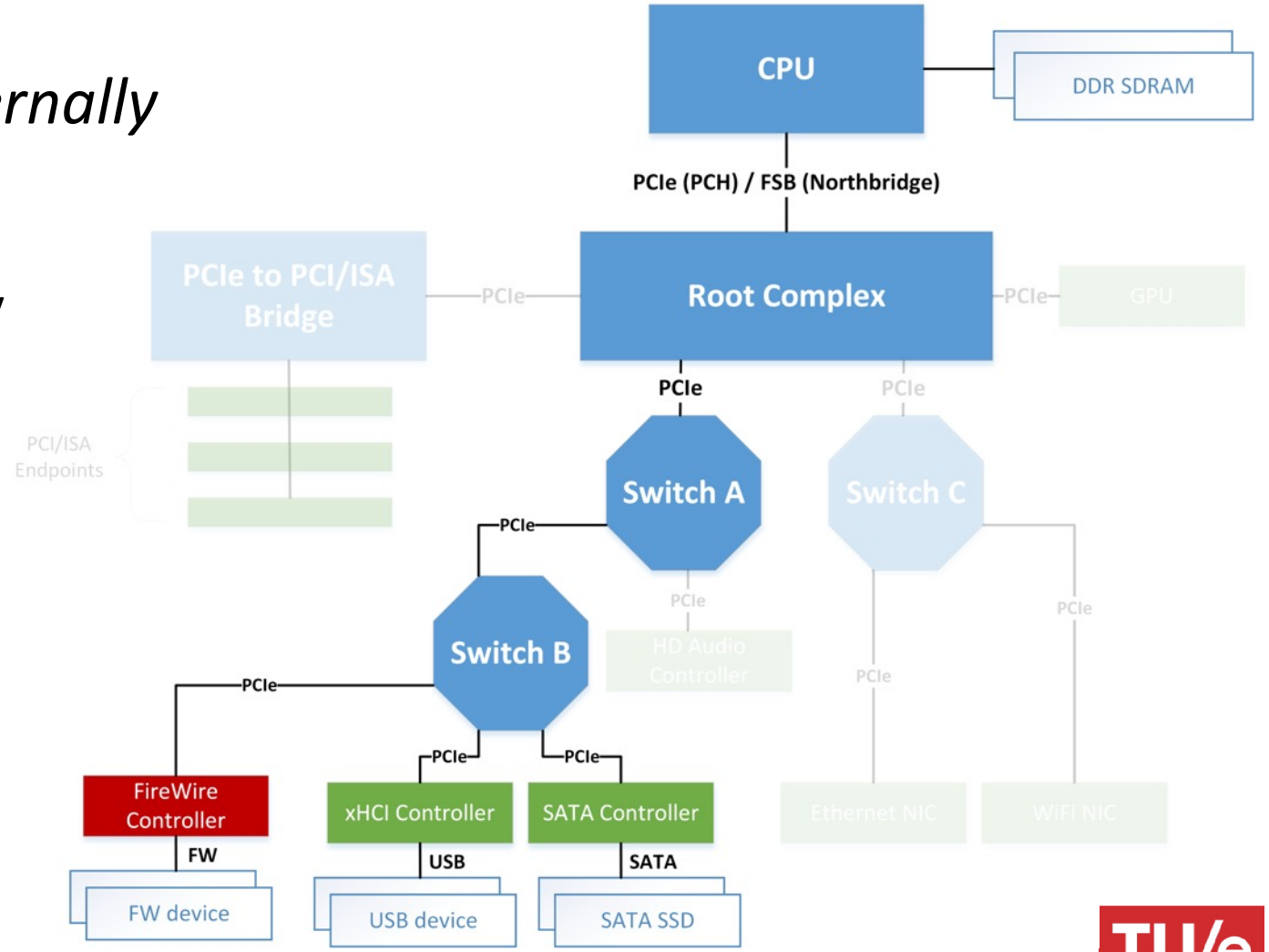


Evolution of PCI Express: Use Cases

- Recent developments deprecate legacy stack in favor of “bare-metal” PCIe
- Why?
  - Facilitate use cases that need more bandwidth and *lower latencies*
  - Reduce load on CPU and system memory

Evolution of PCI Express: Externally

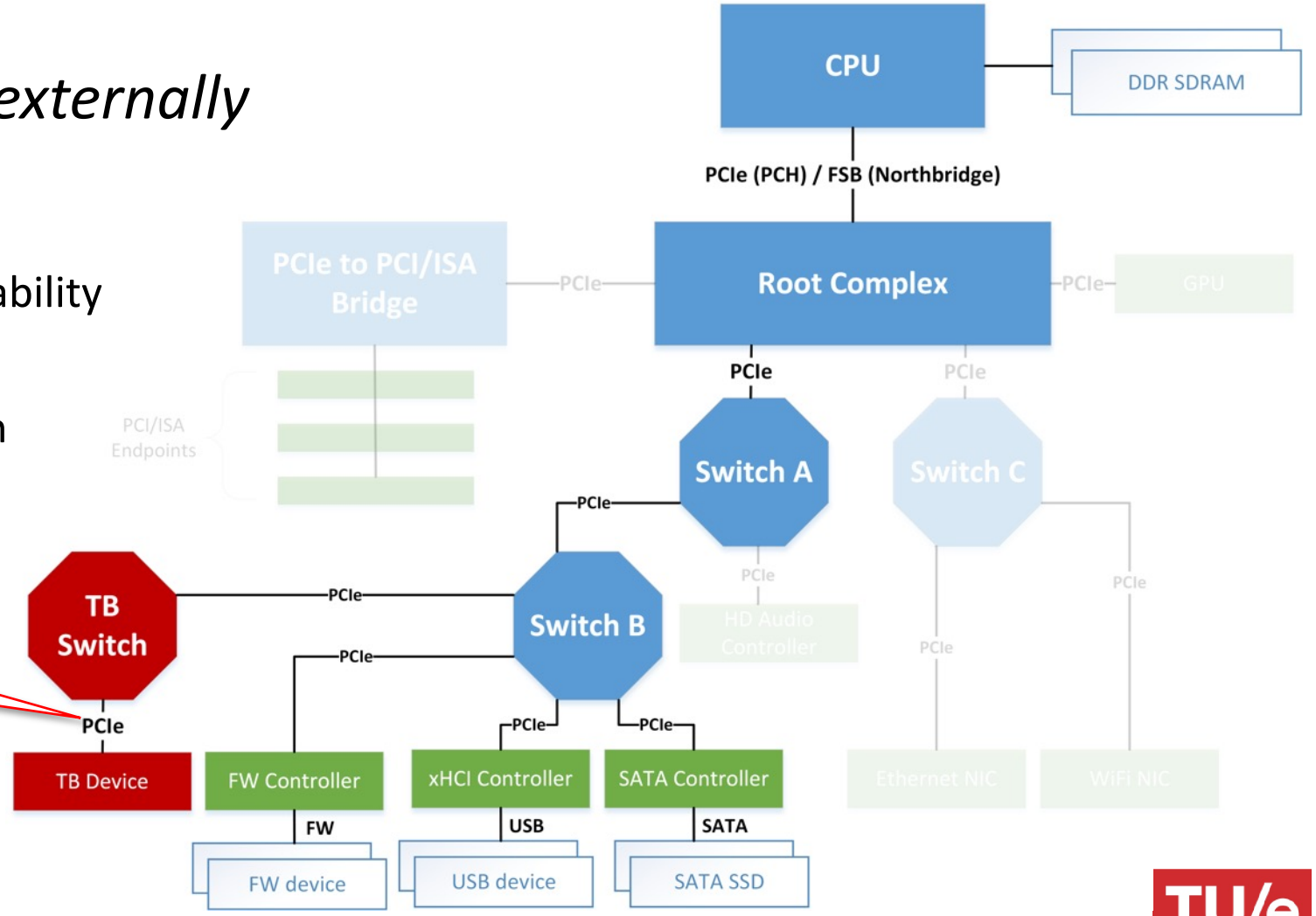
- PCIe is everywhere, even *externally*
  - *FireWire*
    - Abstracts end devices,
      but exposes host DMA capability



Evolution of PCI Express: Externally

- PCIe is everywhere, even *externally*
  - *FireWire*
    - Abstracts end devices,
      but exposes host DMA capability
  - *Thunderbolt*
    - Exposes entire PCIe domain
      to end devices

No abstraction – device attached directly to PCIe

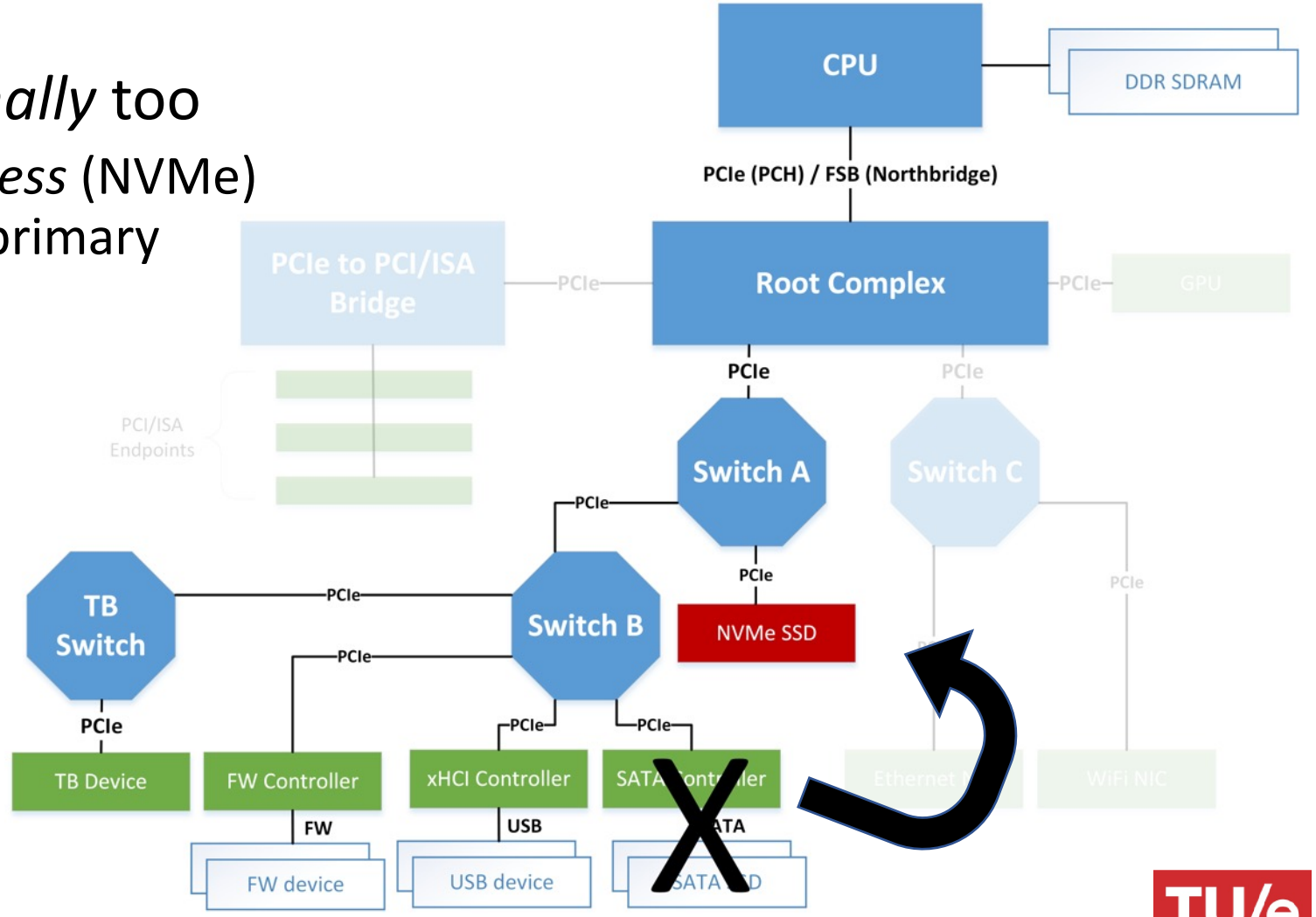


THUNDERBOLT™

TU/e

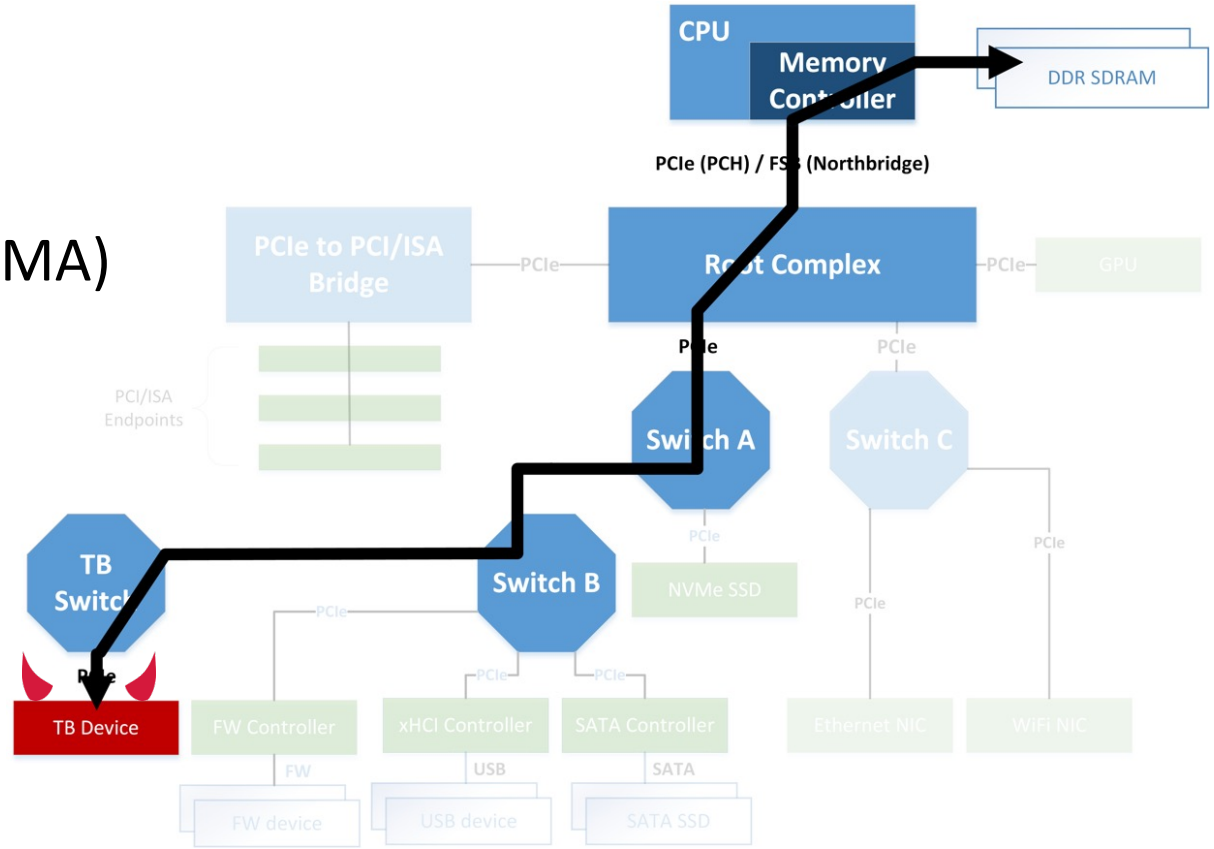
Evolution of PCI Express: Internally

- PCIe is everywhere, *internally* too
  - *Non-Volatile Memory express* (NVMe) bound to replace SATA as primary storage interface



DMA attacks

- **Thunderbolt 1**: no protection against physical attacks
- Plug in malicious device
  → Unrestricted R/W memory access (DMA)
- Access data from encrypted drives
- Persistent access possible, by e.g. installing rootkit



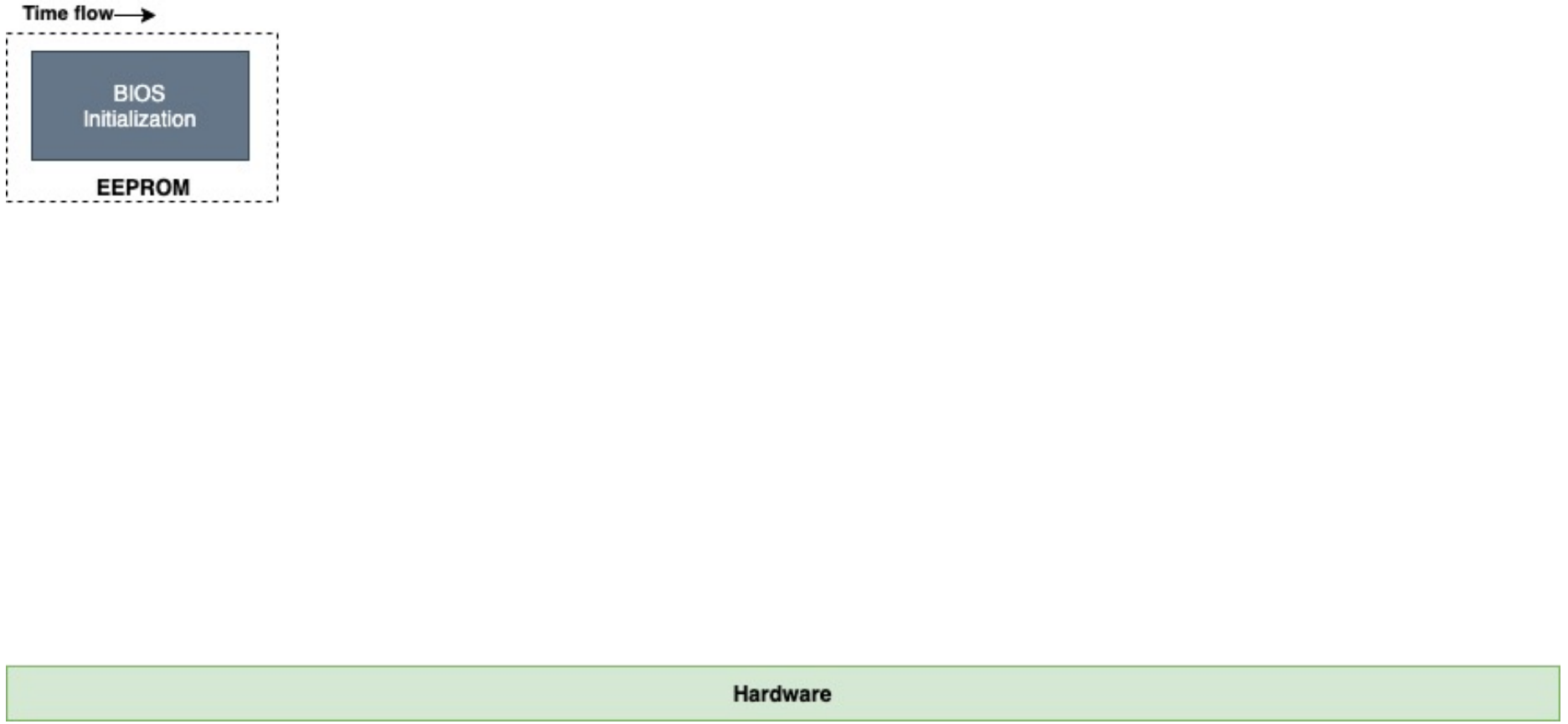
DMA attacks (selected)

- **Owned by an iPod [Dornseif 2004]**
  - First research to demonstrate practical DMA attack
  - Malicious FireWire (FW) device presents Serial Bus Protocol 2 (SPB-2) endpoint, which triggers host controller to allocate DMA channel for fast bulk data transfers
  - Several authors release exploitation tools [Boileau 2006] [Plegdon 2007]
  - Improved upon for memory forensics [Witherden 2010]
  - "Improved upon" in law enforcement spyware such as FinFireWire [Gamma 2011]
- **Subverting Windows 7 x64 kernel with DMA attacks [Aumaitre 2009]**
  - First PCI-based attack through custom PCI device with DMA engine
- **Inception [Maartmann-Moe 2014]**
  - Improves upon Witherden's `libforensic1394` by presenting virtual SBP-2 interface through ExpressCard, FW device + Thunderbolt-to-FW adapter
- **PCILeech [Frisk 2016]**
  - First native PCIe attack
  - DMA attack using FPGA with PCIe PHY (full size, ExpressCard, miniPCIe, M.2-NVMe), optionally tunneled through Thunderbolt enclosure
  - Improved later with various functionality: e.g. dumping FDE keys, dumping UEFI memory regions, patching Windows lock screen process
- **Thunderclap [Markettos et al. 2019]**
  - Replaces PCIe endpoint in TB device with malicious one, then performs DMA attack
  - Does not break Security Levels access control, but relies on tricking user into authorizing malicious device
- **Thunderspy [Ruytenberg 2020]**
  - First research to demonstrate full system compromise by breaking Thunderbolt security, including Security Levels
  - Collection of seven critical vulnerabilities and nine exploitation scenarios affecting Thunderbolt 1, 2 and 3
  - Details in next lecture

TU/e

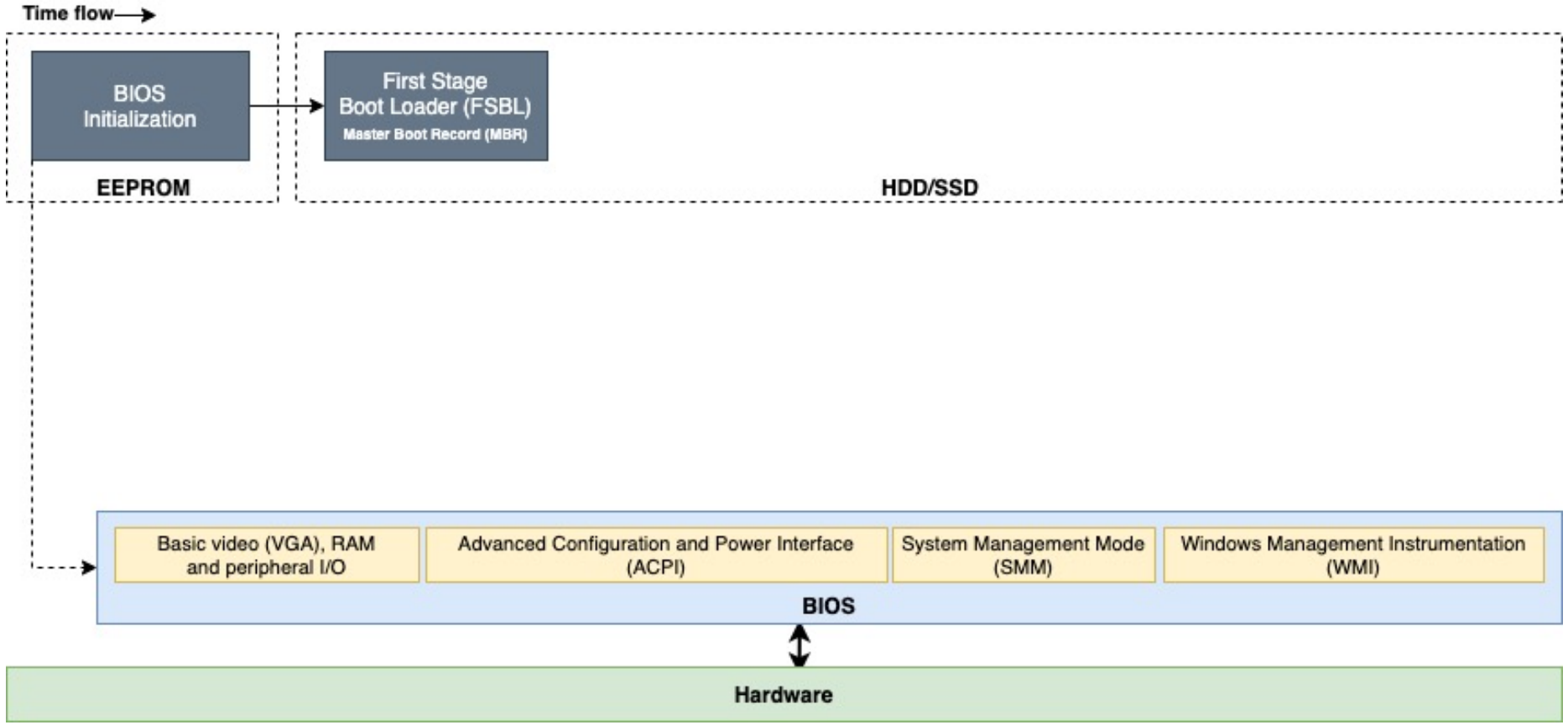
The x86 Boot Process

Legacy Boot



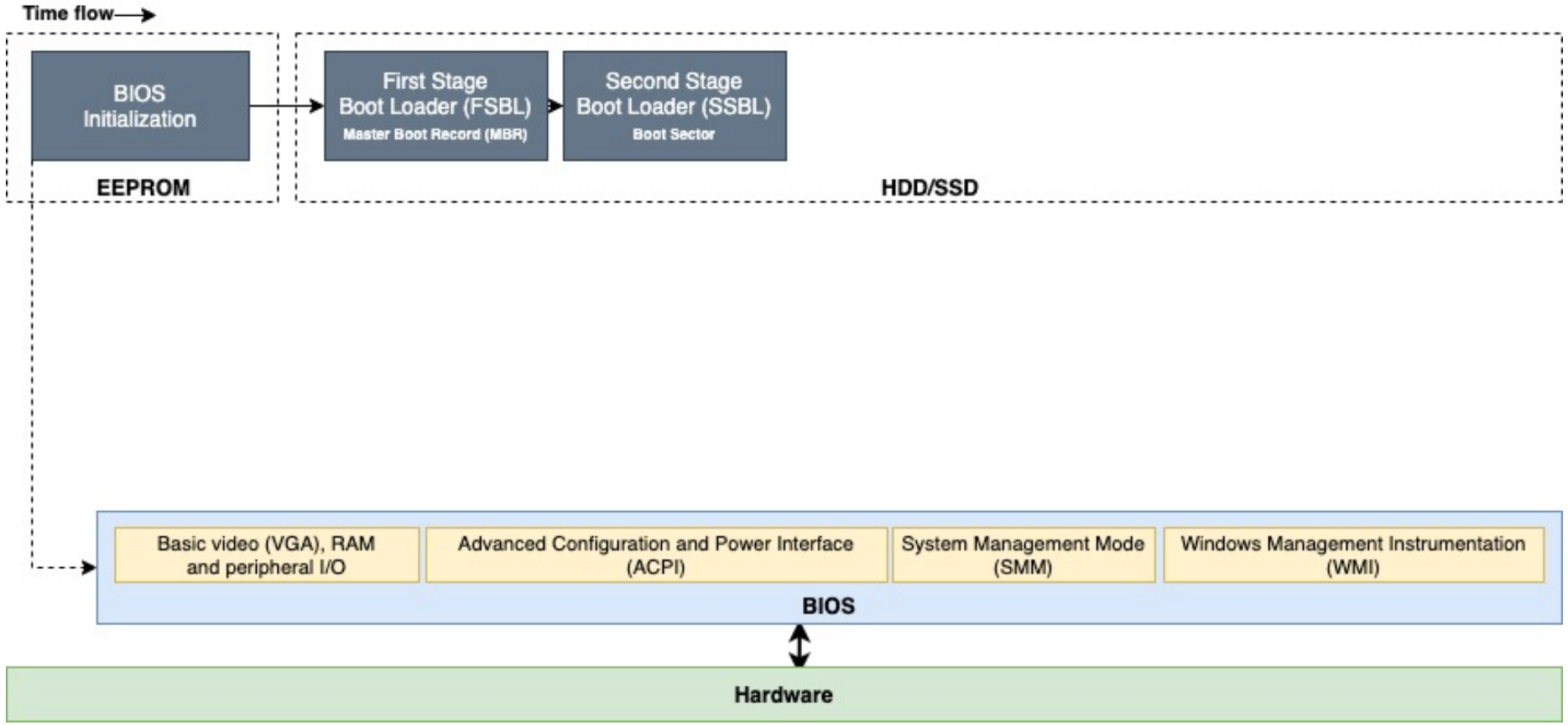
- CPU is powered on, loads Basic Input/Output System (BIOS)

Legacy Boot



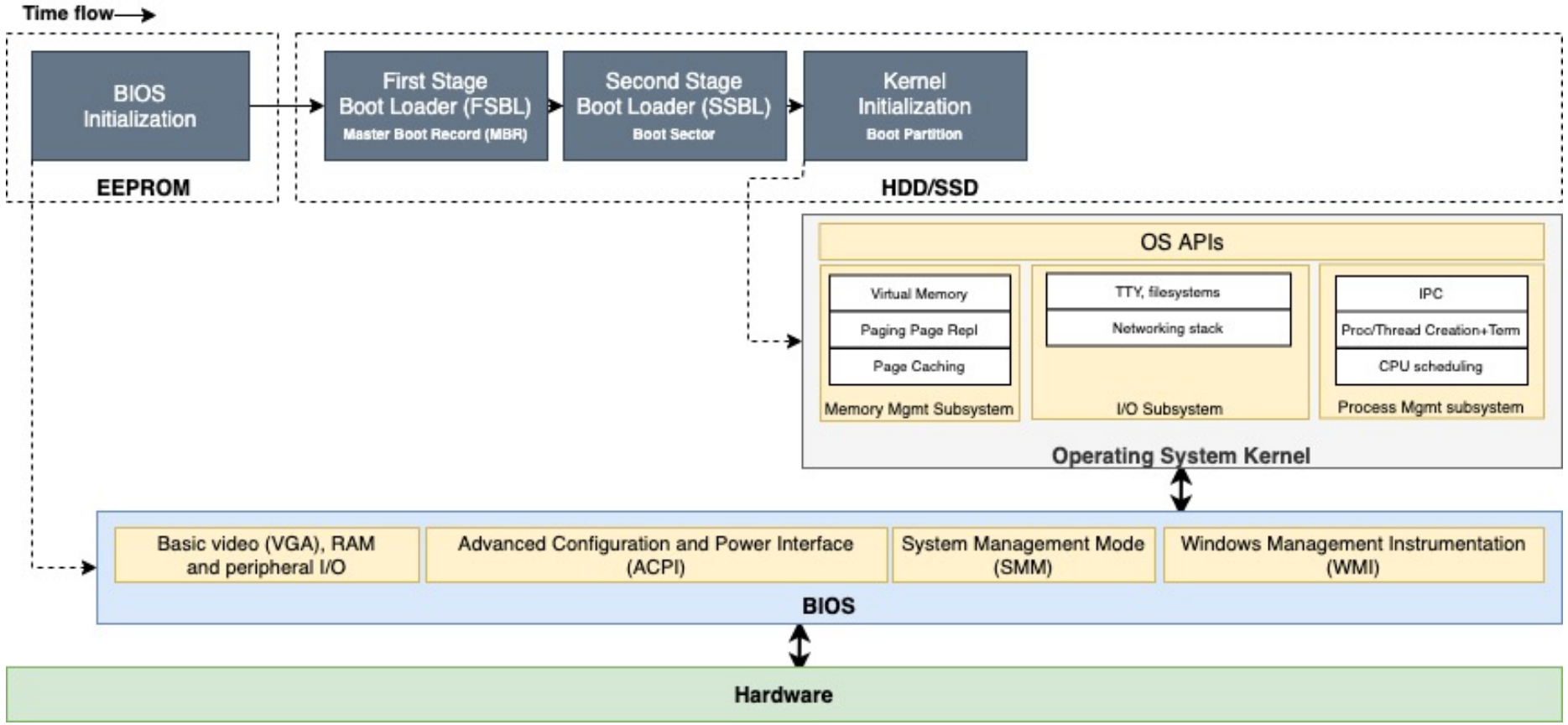
- BIOS
  - Performs basic hardware initialization (e.g. RAM, GPU, storage) and testing (Power On Self Test)
  - Enumerates hardware capabilities and provides basic abstractions to OS, including ACPI, SMM, WMI
  - Loads Master Boot Record (MBR), executes embedded First Stage Boot Loader (e.g. GRUB/NTLDR)

Legacy Boot



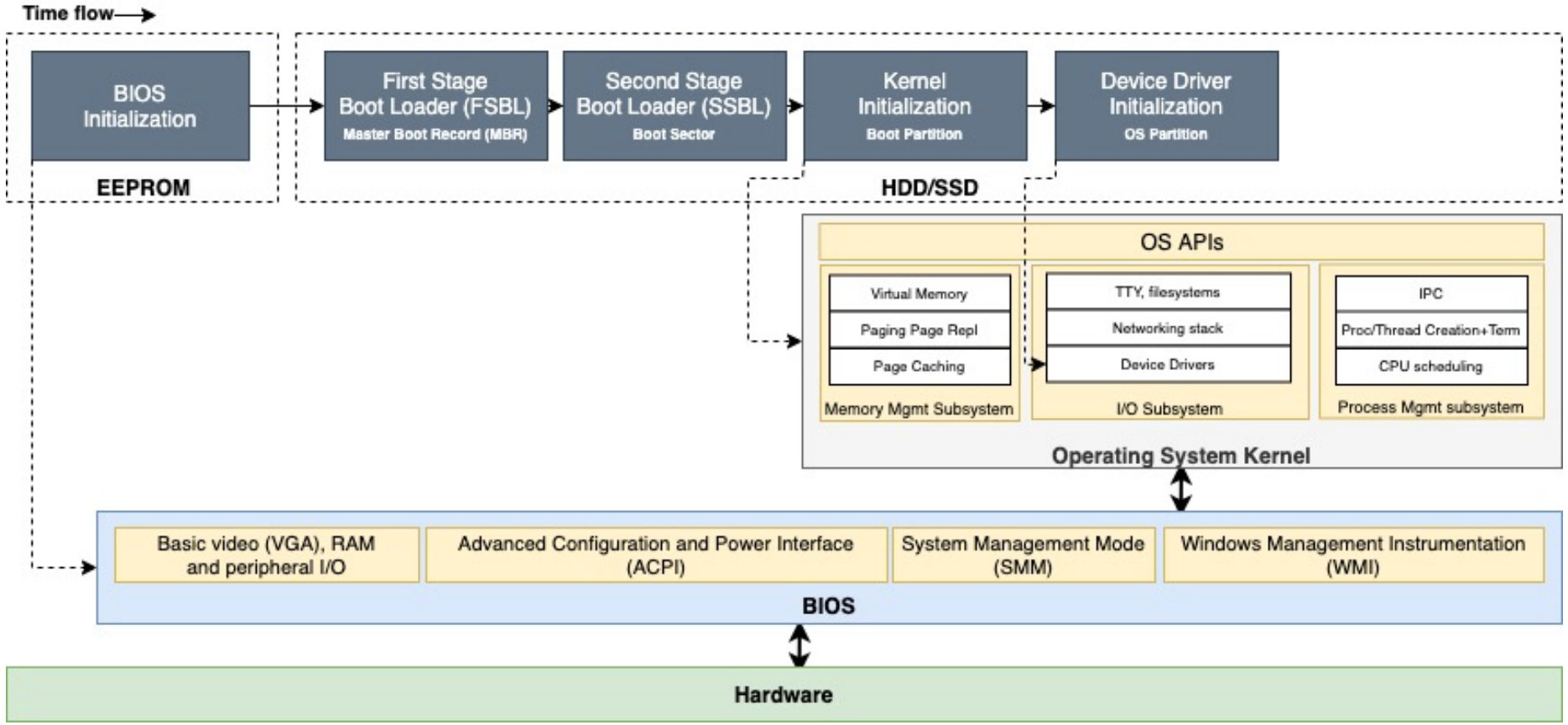
- First Stage Boot Loader
  - Mounts filesystem on first bootable partition
  - Loads SSBL into RAM (e.g. second stage GRUB/NTLDR)

Legacy Boot



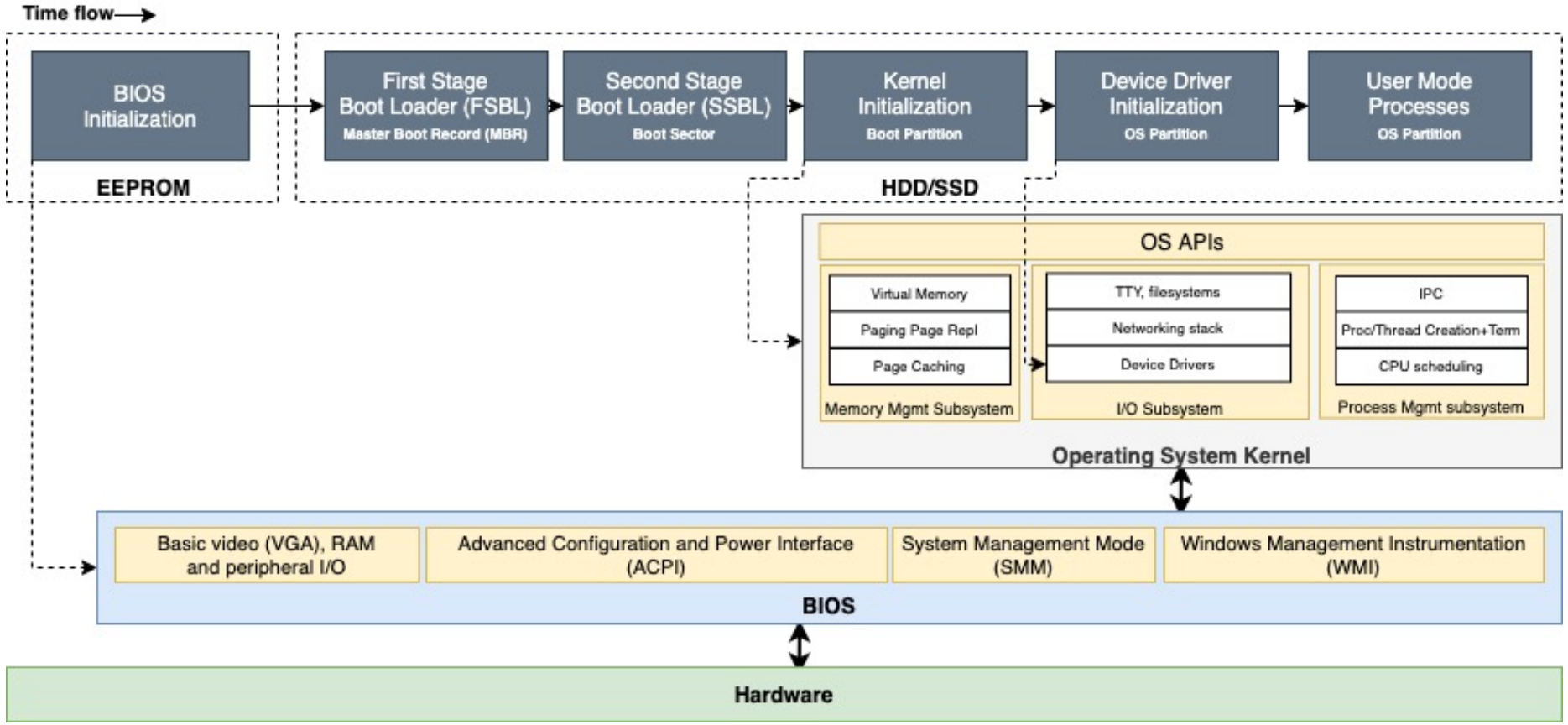
- Second Stage Boot Loader
  - Mounts OS filesystem, loads kernel into RAM, executes kernel

Legacy Boot



- Operating System kernel
  - Initializes memory management, I/O, process management subsystems
  - Loads device drivers, enabling full peripheral functionality

Legacy Boot



- Operating System kernel
  - Launches user mode processes

Legacy Boot – Limitations

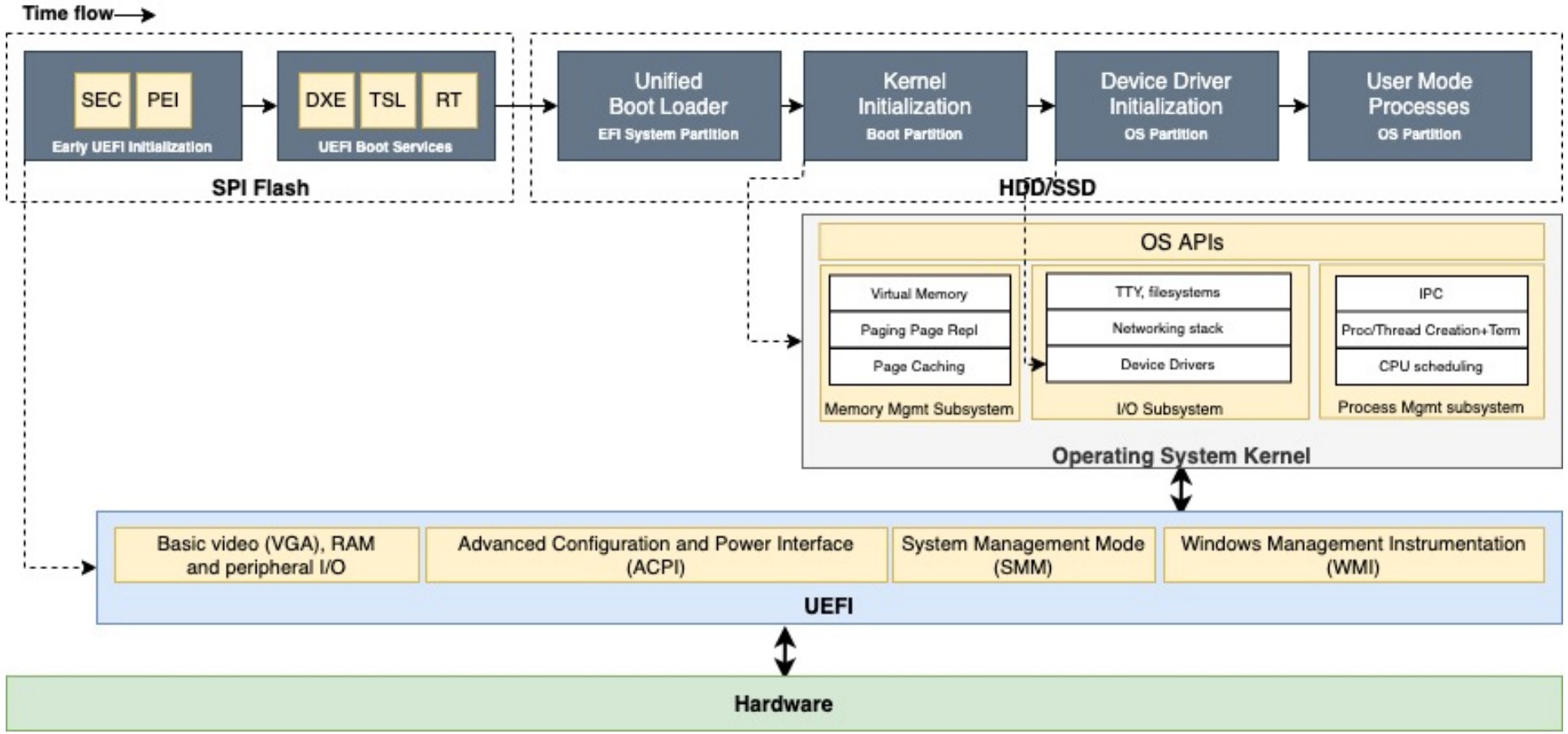
- First BIOS implementations date from 1970's
- Legacy technology renders code base difficult to maintain
  - 16-bit "real mode" x86 assembly
  - Typically limited to 1 MB EEPROM
  - MBR imposes inflexible and resource-constrained boot process
    - FSBL limited to 440 bytes, need to resort to SSBL for further system initialization
    - Partition table limited to 4 partitions
    - Uses 32-bits for logical block addressing, limiting storage to 2 TB (16 TB using "Advanced Format")
  - Monolithic architecture complicates adding new hardware support
  - Single-threaded code preventing parallelizing hardware initialization, leading to extended boot times
  - **No security:** boot process can be hijacked by malicious code

UEFI Boot

Unified Extensible Firmware Interface (UEFI)

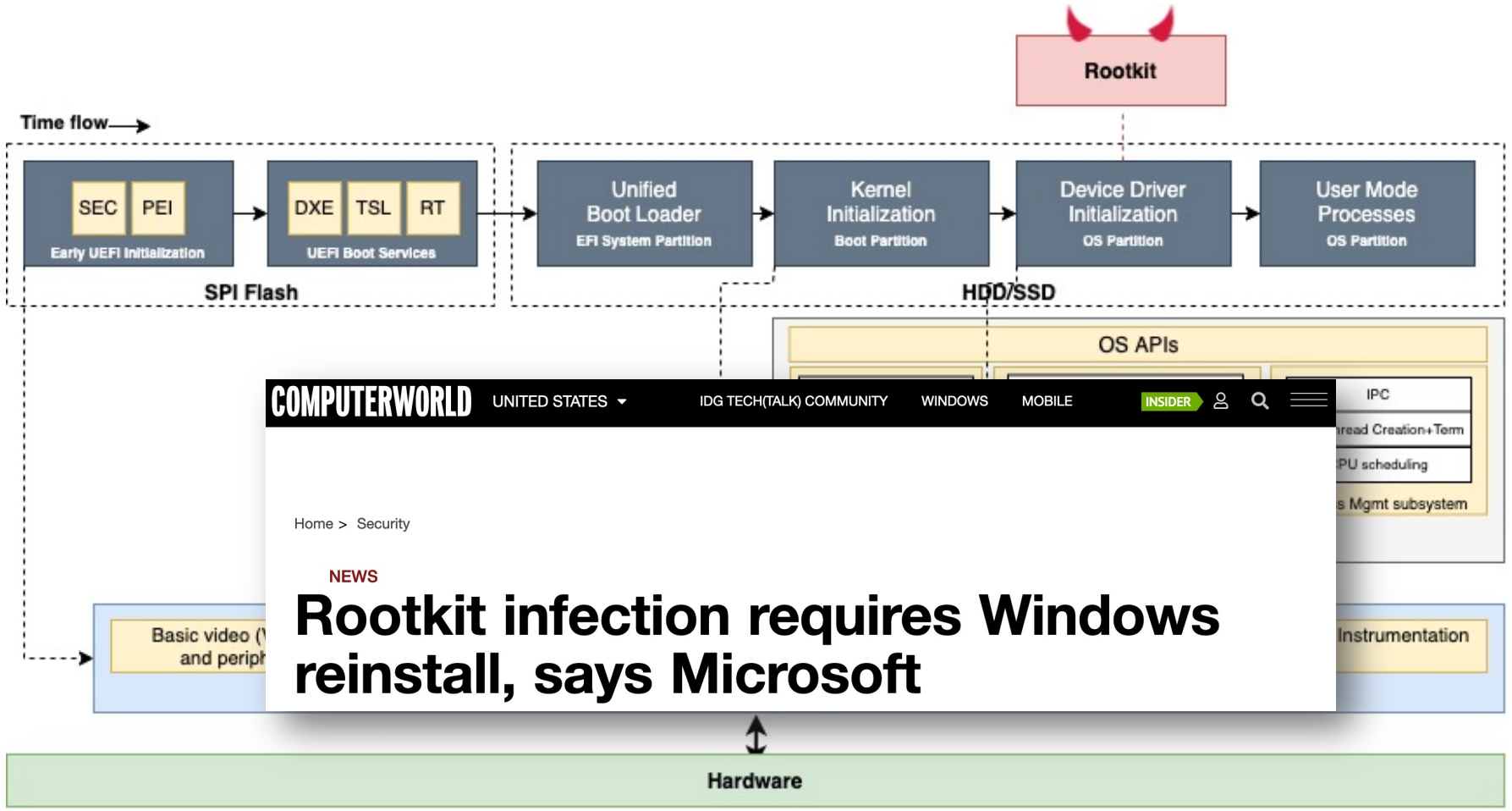
- Early versions developed exclusively by Intel (2000). Contributed to UEFI Forum and subsequently adopted by
  - Apple, for Intel Macs (2006 – today)
  - x86 OEM/ODMs (2011 – today)
  - Recent ARM and PowerPC implementations available, but not commonly used
- Stored on SPI flash, typically up to 64 MB
- GUID Partition Table (GPT) boot alleviates most MBR limitations
  - Unified boot loader binary stored on FAT32-formatted EFI System Partition (ESP)
  - Partition table enables up to 128 partitions
  - Uses 64 bits for logic block addressing – storage addressable up to 9.4 ZB
- Boot stages
  - Security Phase (SEC): Minimal assembly to initialize microcode and co-processors (Intel ME, AMD PSP, TPM)
  - Pre-EFI Initialization (PEI): Basic hardware initialization + POST, ACPI sleep/resume handling
  - Driver Execution Environment (DXE): initialize boot time device drivers
  - Transient System Load (TSL): Load user-selected boot loader from ESP
  - Runtime (RT): UEFI hands over control to OS boot loader
- Note: often still referred to as “BIOS” in vendor documentation, literature

UEFI Boot

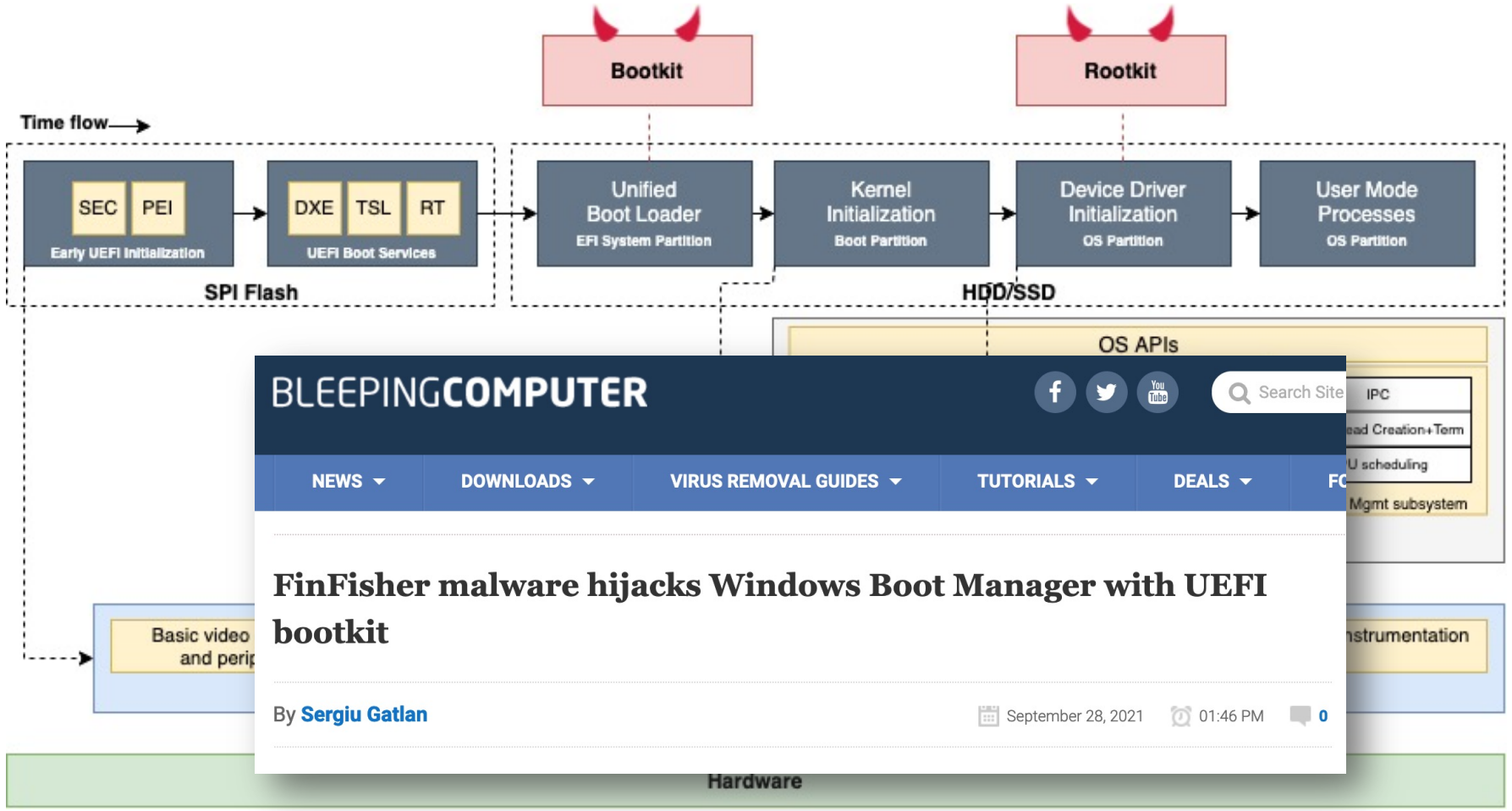


What about security?

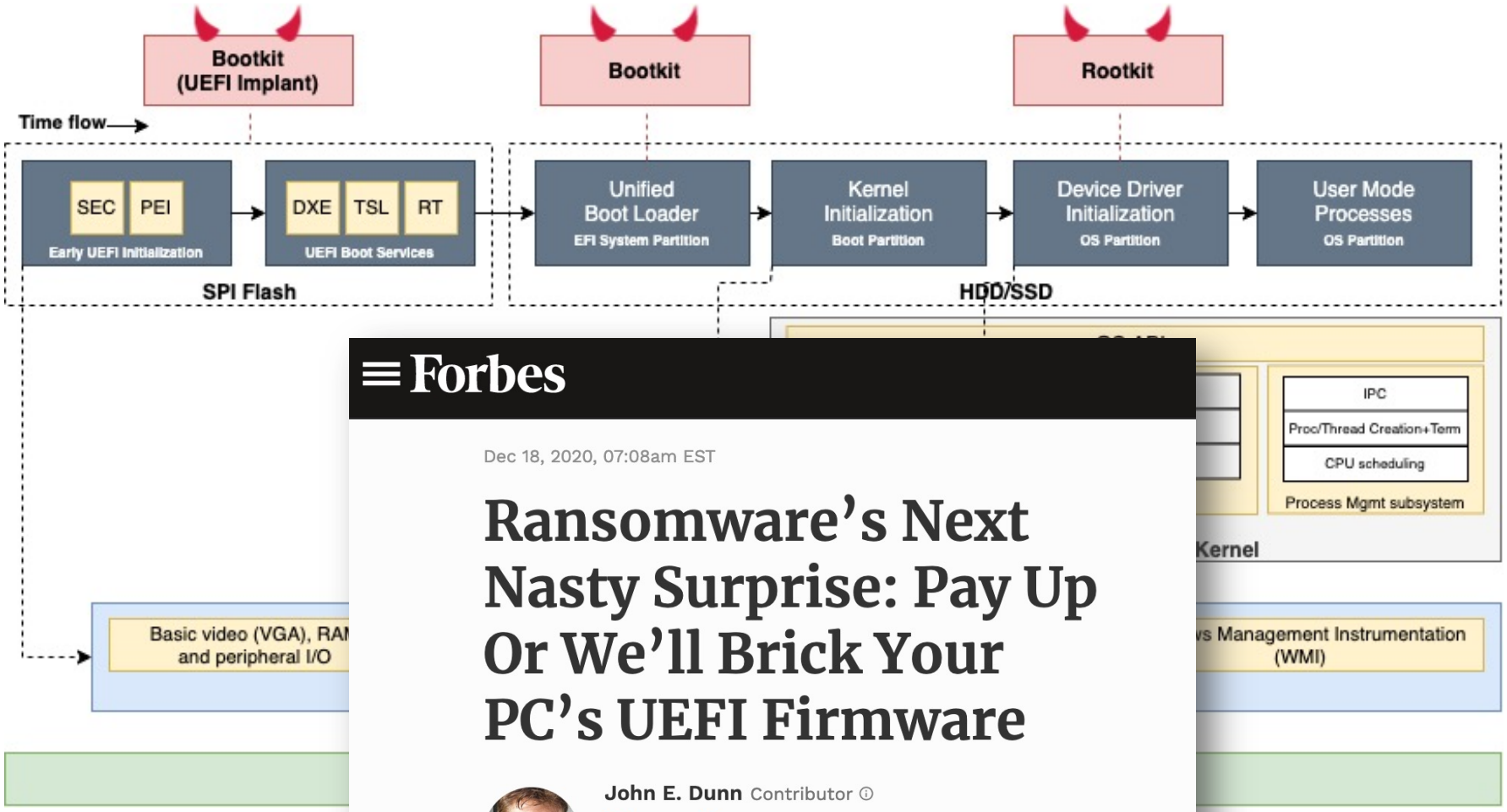
UEFI Boot – Attack Vectors



UEFI Boot – Attack Vectors



UEFI Boot – Attack Vectors



Forbes

Dec 18, 2020, 07:08am EST

Ransomware's Next Nasty Surprise: Pay Up Or We'll Brick Your PC's UEFI Firmware

John E. Dunn Contributor ⓘ

Cybersecurity

Trying to explain cybersecurity woes and why they matter

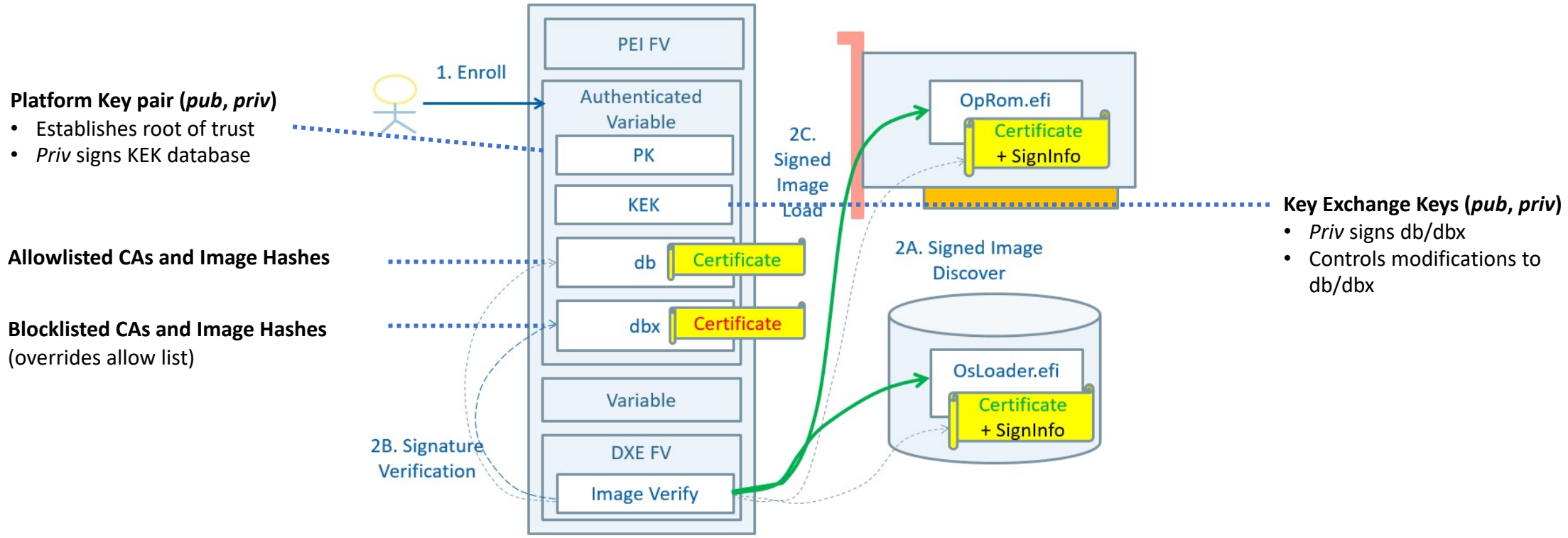
TU/e

Addressing UEFI Boot Attack Vectors

- **Verified Boot / Boot Guard**
  - Protects against malicious firmware implants
  - Cryptographically verifies UEFI integrity
- **Driver signing**
  - Ensures driver authenticity (originates from device vendor) and integrity (tamper-resistance)
  - OS vendors require device vendors to submit drivers for evaluation
    - Linux (kernel.org): submit code for functional + security review
    - Windows: semi-blackbox testing through WHQL program
  - Upon passing verification
    - Linux: distribution vendors sign pre-compiled driver binaries
    - Windows: Microsoft signs driver's commercial release certificate; device vendor signs driver using private key to latter certificate
- **Secure Boot**
  - Protects against malicious, unsigned code early in boot process
  - Cryptographically verifies boot chain: OS bootloader, kernel, drivers

Secure Boot: A Closer Look

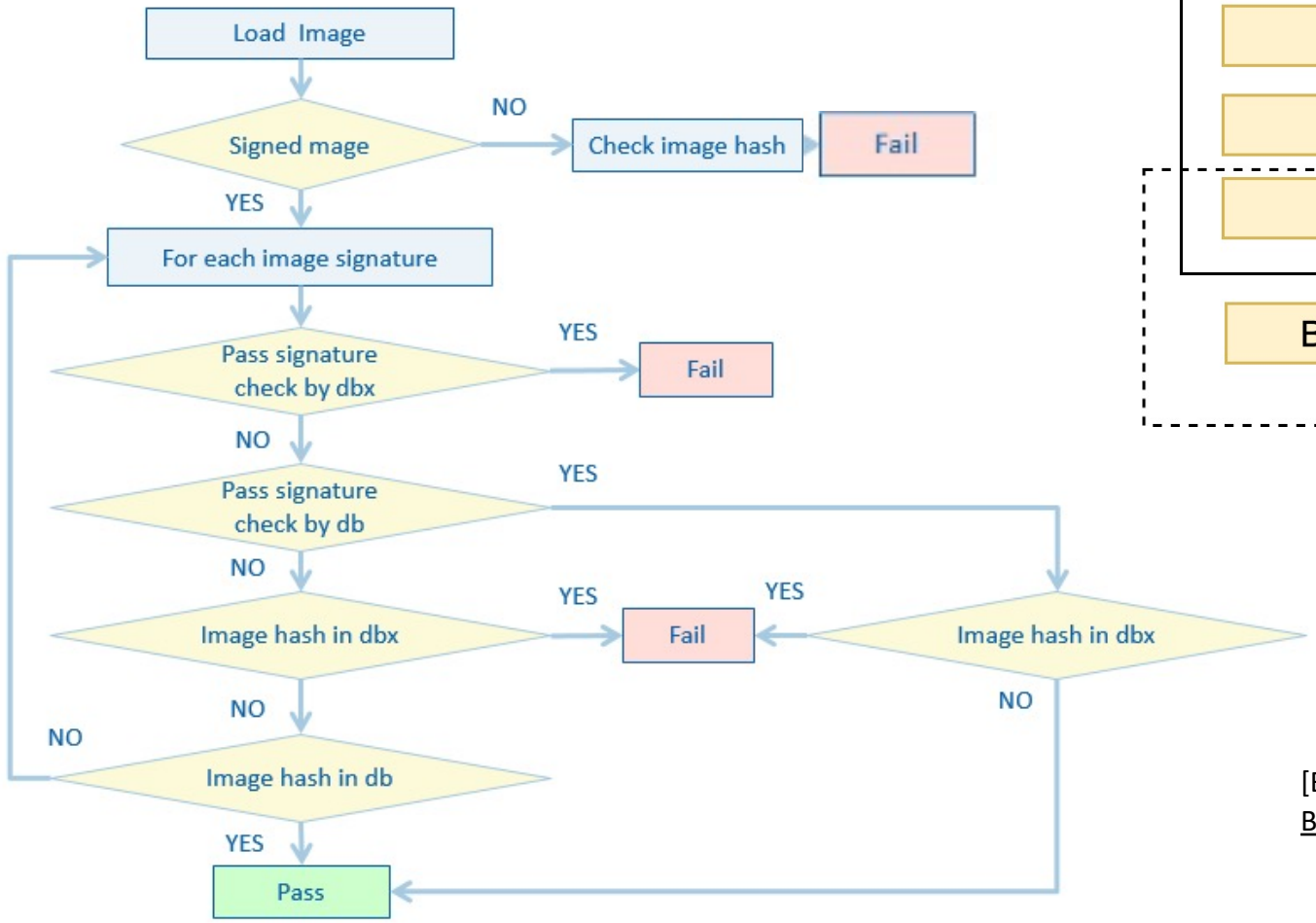
- Protects against malicious, unsigned code early in boot process
- Cryptographically verifies boot chain: OS bootloader, kernel, drivers

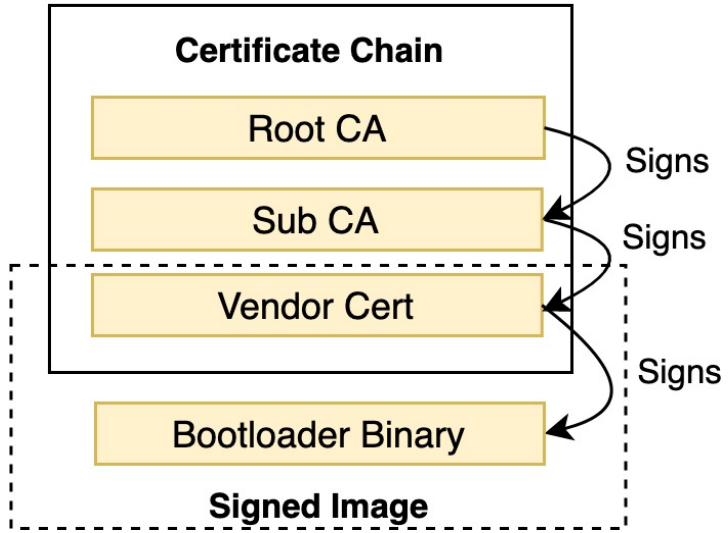


[Based on "Understanding the UEFI Secure Boot Chain" – TianoCore/EDK2]

Secure Boot: A Closer Look

- Protects against malicious, unsigned code early in boot process
- Cryptographically verifies boot chain: **OS bootloader**, kernel, drivers

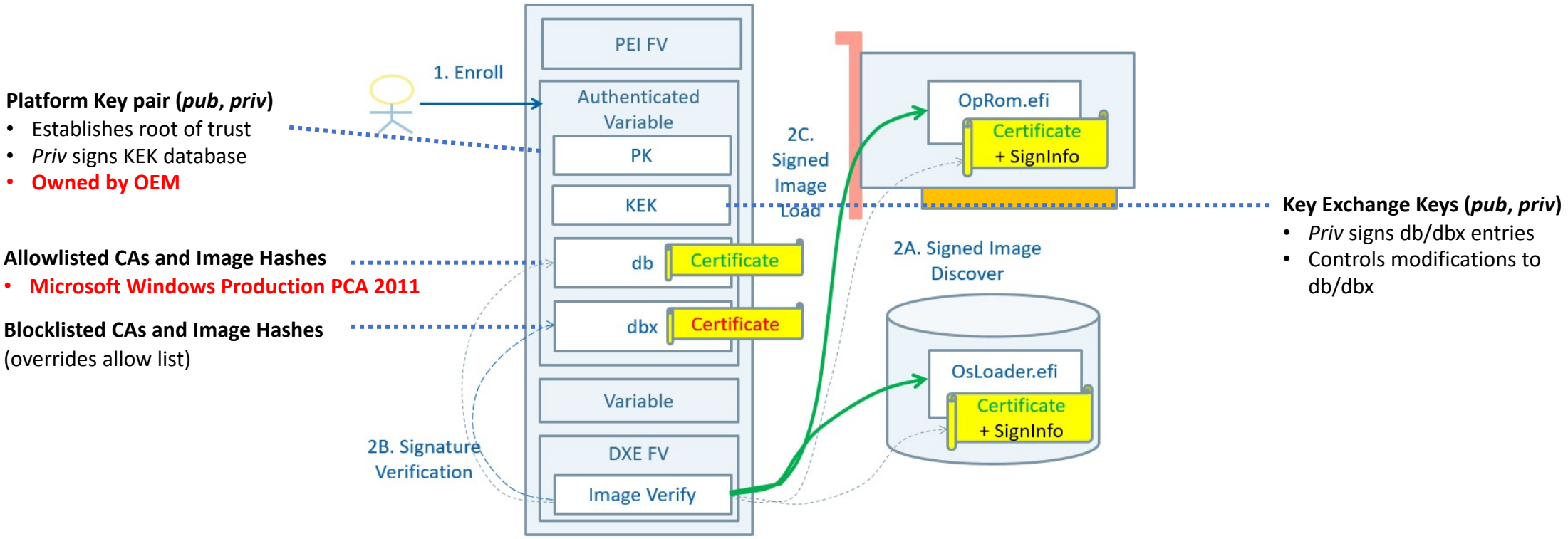




[Based on "Understanding the UEFI Secure Boot Chain" – TianoCore/EDK2]

Secure Boot: A Closer Look

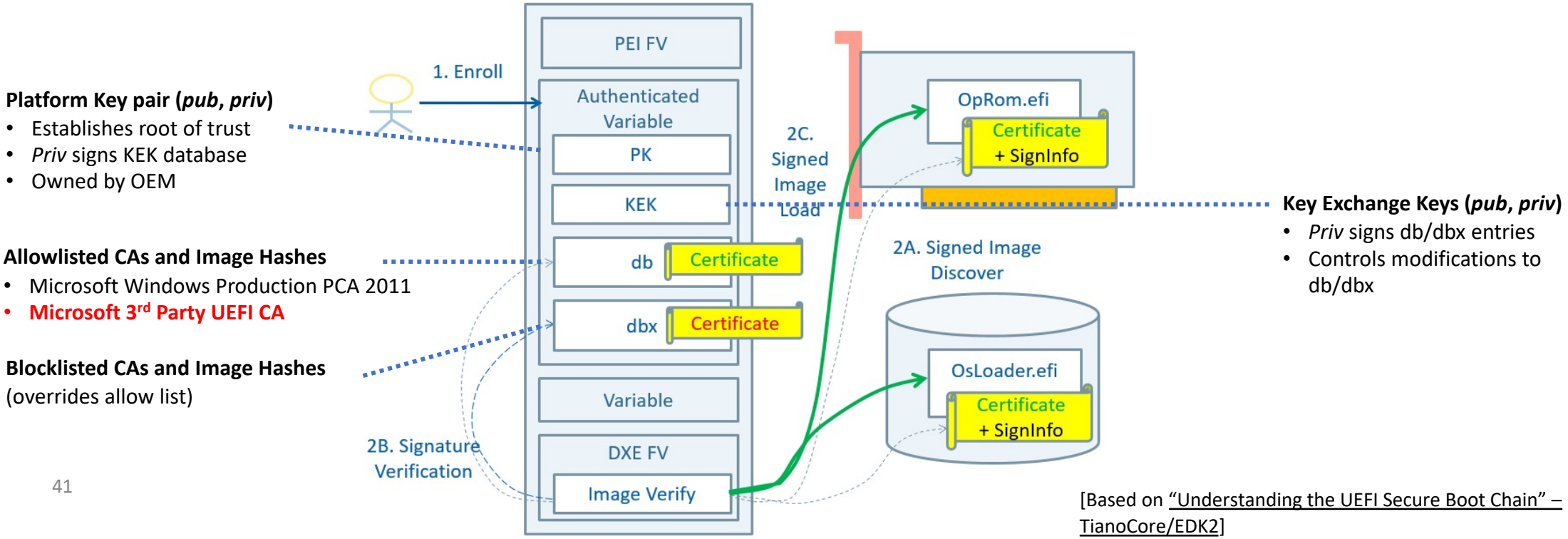
- Secure Boot trust model similar to PKI, but with significant exceptions
  - OEM-controlled PK; Root CAs exclusively controlled by Microsoft
- Raises concerns on Secure Boot chain ownership



[Based on "Understanding the UEFI Secure Boot Chain" – TianoCore/EDK2]

Secure Boot: A Closer Look

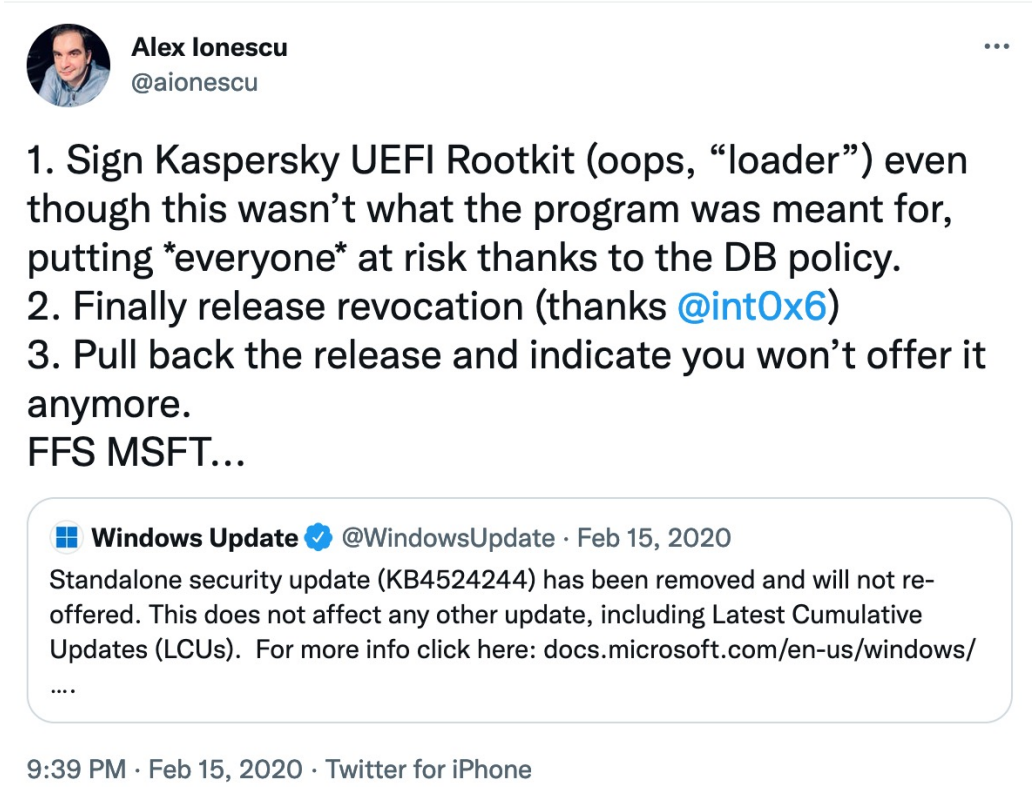
- Recent UEFI implementations enable user-replaceable PK
- Secondary Microsoft Root CA signs third party boot loaders
  - Red Hat: SHIM first stage boot loader
    - Used to chainload e.g. GRUB2, systemd-boot, EFI Stub
    - Enables users to self-sign boot loaders + drivers using “Machine Owner Key” (MOK)
  - Anti-malware and imaging solutions, e.g. live USB flash drive environment



[Based on “Understanding the UEFI Secure Boot Chain” – TianoCore/EDK2]

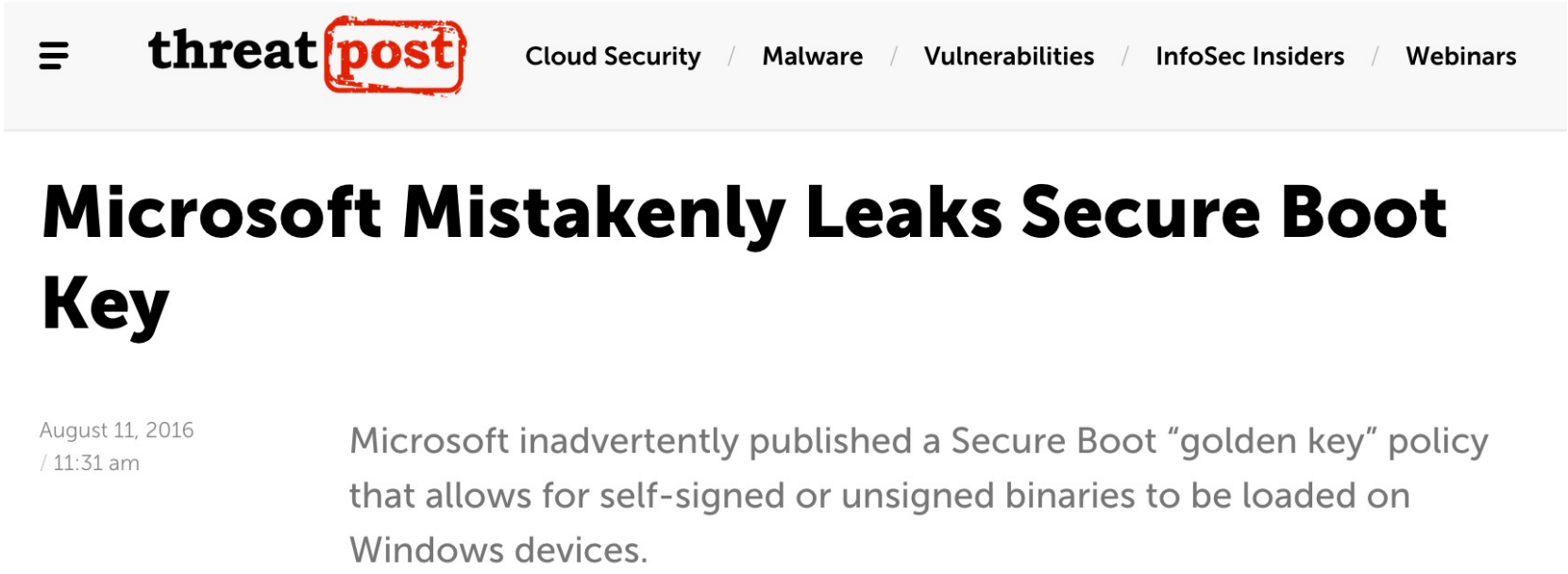
Secure Boot: How Secure is It?

- Boot chain security relies on
  - Root and subordinate CAs ensuring appropriate key management practices
  - Root and subordinate CAs signing only trustworthy binaries



Secure Boot: How Secure is It?

- Boot chain security relies on
  - Root and subordinate CAs ensuring appropriate key management practices
  - Root and subordinate CAs signing only trustworthy binaries
  - Signed binaries verifying any subsequently chainloaded binaries



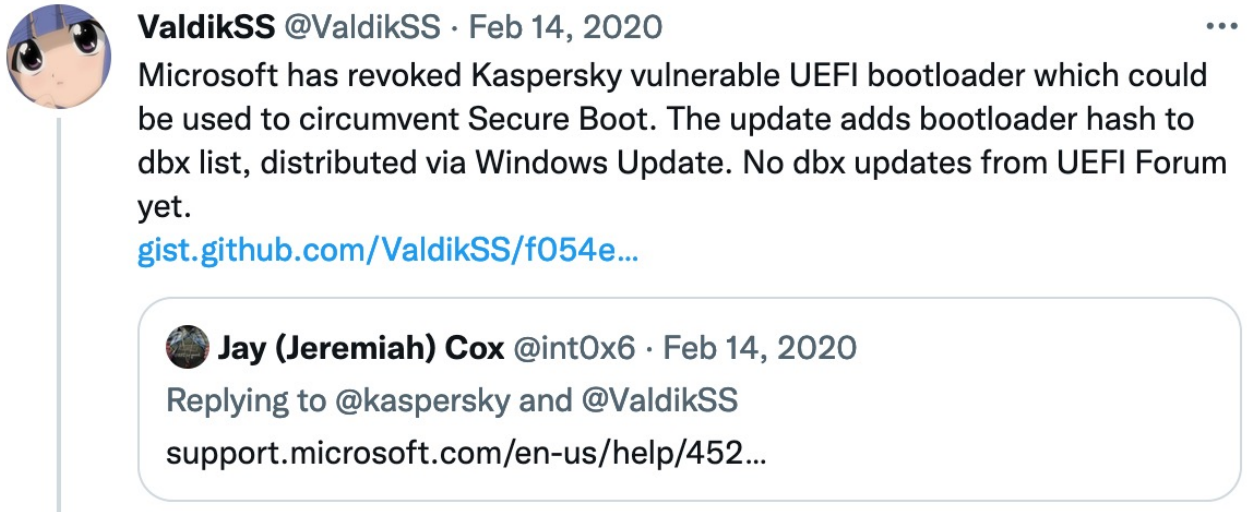
Secure Boot: How Secure is It?

- Boot chain security relies on
  - Root and subordinate CAs ensuring appropriate key management practices
  - Root and subordinate CAs signing only trustworthy binaries
  - Signed binaries verifying any subsequently chainloaded binaries
  - Signed binaries not introducing vulnerabilities of their own

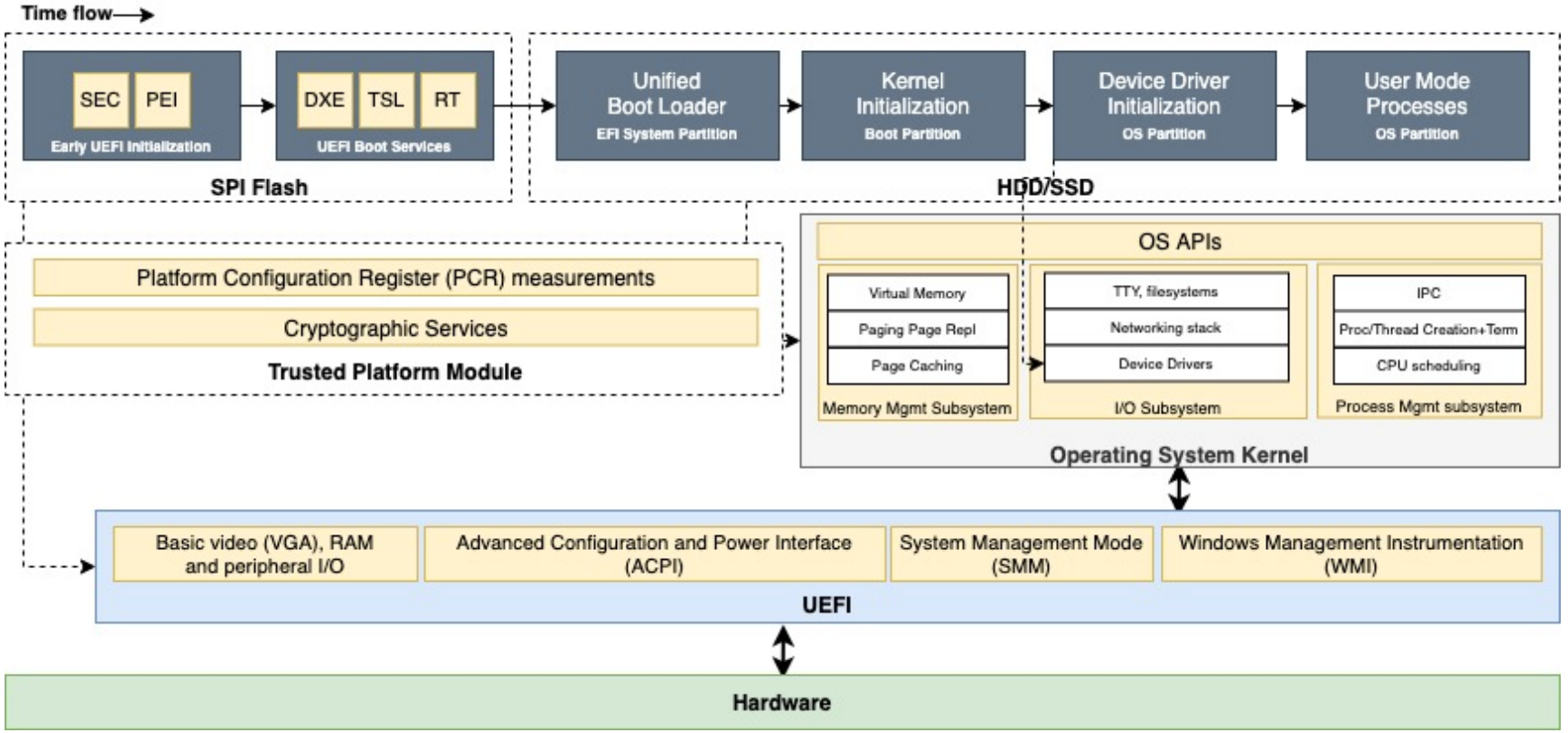
"BootHole" vulnerability in the GRUB2 bootloader opens up Windows and Linux devices using Secure Boot to attack. All operating systems using GRUB2 with Secure Boot must release new installers and bootloaders.

Secure Boot: How Secure is It?

- Boot chain security relies on
  - Root and subordinate CAs ensuring appropriate key management practices
  - Root and subordinate CAs signing only trustworthy binaries
  - Signed binaries verifying any subsequently chainloaded binaries
  - Signed binaries not introducing vulnerabilities of their own
  - If all fails, CAs timely identifying affected public keys/binaries; OEMs distributing dbx updates

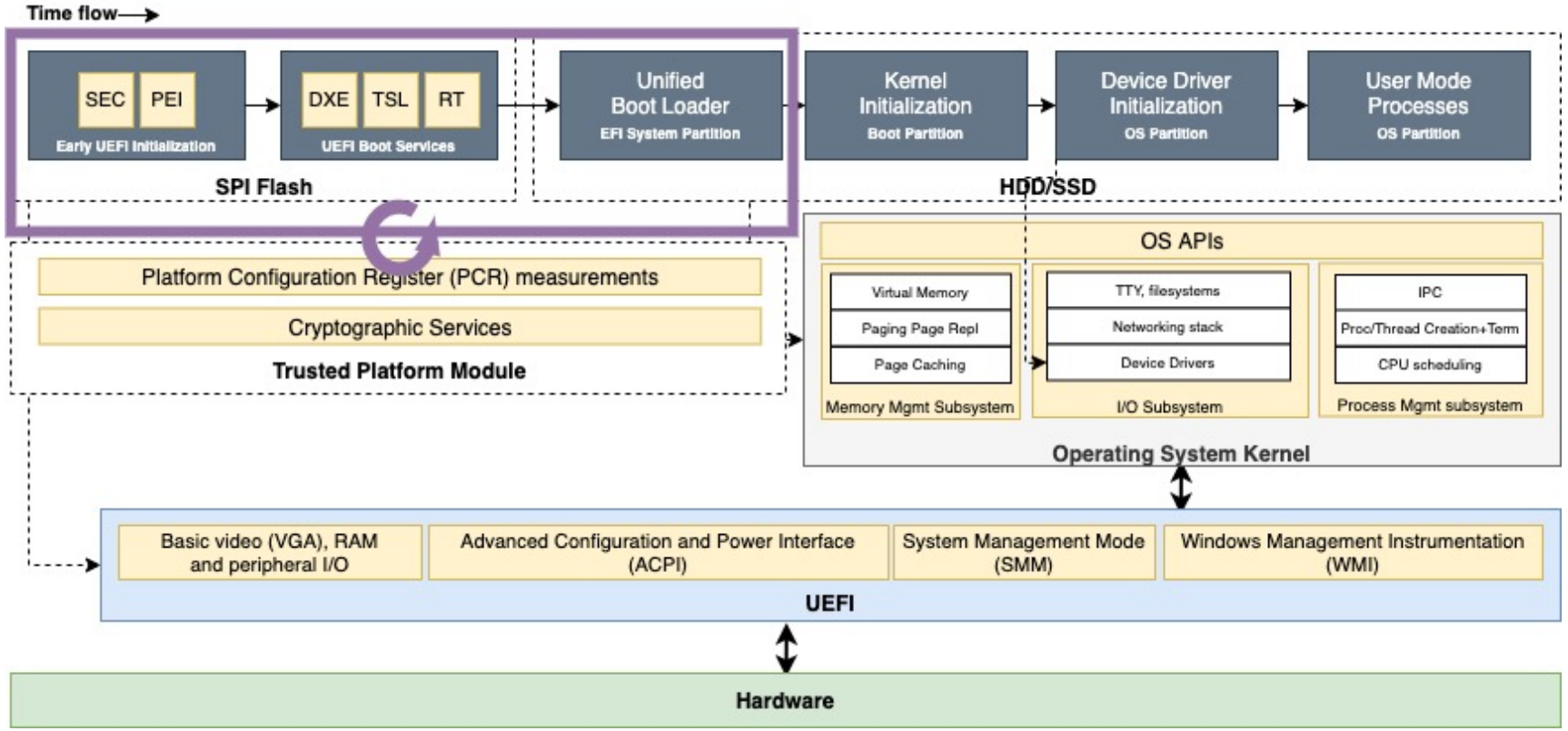


Measured Boot



- Requires autonomous, distinct agent that continuously monitors boot process

Measured Boot



- Typical x86 approach employs TPM
  - Learns about known good state, i.e. “one-time” setup that computes hash over UEFI image, UEFI config, boot loader image.
  - Stores state as PCR tuple. Continuously computes PCR on each boot cycle – PCR changes ⇔ boot chain modified

References

Obligatory reading

- Unified Extensible Firmware Interface Specification v2.9 (2021)
  - Chapter 32 – Secure Boot

Background reading

- Andrew S. Tanenbaum and Herbert Bos, “Modern Operating Systems” (2014)
  - Chapter 1 – Introduction, 5 – Input/Output, 9 – Security
- Trusted Computing Group, “Using the TPM to Solve Today’s Most Urgent Cybersecurity Problems” (2014)
  - Slides 18 - 28
- Trusted Computing Group, “Trusted Platform Module 2.0 Library – Part 1: Architecture” (2019)
  - Chapter 11 – TPM Architecture

Admin

- **Next lecture**
  - Case study: Thunderspy
- **Quiz**
  - Verify your understanding of material
- **Questions?**
  - Live session on Dec 14
  - Reach out via email