Linux Security Summit NA '23

MPK/ PKS Kernel Compartmentalization

Sebastian Österlund
Offensive Security Researcher – IPAS STORM/ SPEAR

intel®

About Sebastian

Background

- Offensive Security Researcher at Intel IPAS STORM/ SPEAR
- Previously PhD student in Computer and Network Security at the VU Amsterdam (VUSec)
- Previous Research: OS defenses, Speculative Execution Attacks (RIDL/ MDS), Fuzzing, Compilers
- Currently: static analysis of microcode, hardening Operating Systems using new HW features

Why compartmentalize?

- Nowadays much privileged third-party code in kernel
  - Drivers, e
- No need fo accessible
- Beyond software bugs: transient execution attacks

Kernel compartmentalization using new HW features

A single memory error could expose all available memory

Transient execution attacks

Spectre & friends

- Have been quite a headache for the kernel
- Properly mitigating could be challenging:
  - E.g., Core scheduling
- Potential Result: functionality could get disabled (Hyper-Threading, BPF unprivileged mode, …)

```
// x, array2 attacker-controlled
if (x < array1_size) {
     secret = array1[x];
     …
     z = array2[secret * 1024];
     // Mem access depending on secret
     // leaves uarch side-effect
}
```

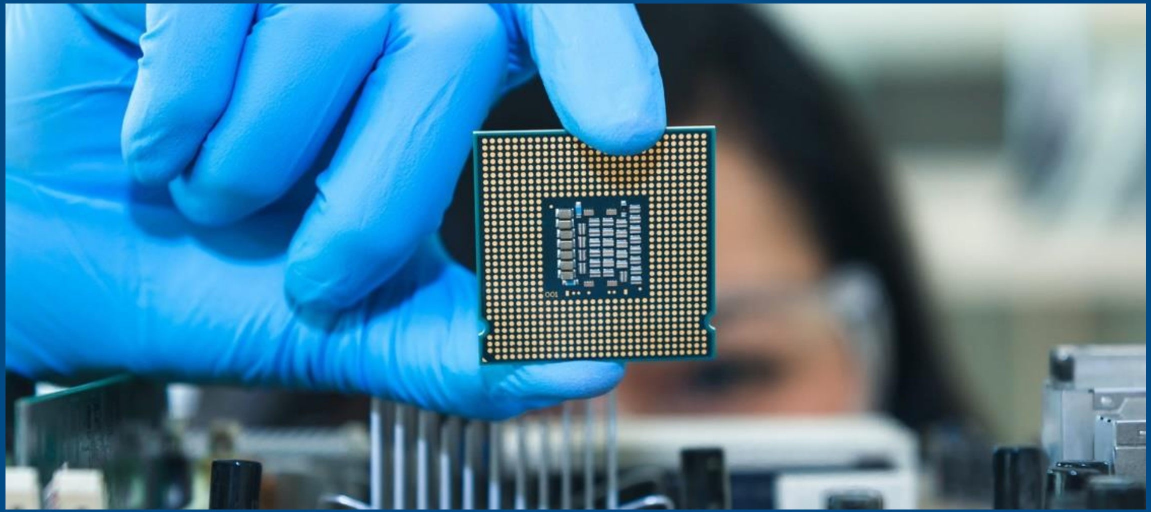
```
// Prepare leak buffer
flush_buffer(buffer);

// Spectre, MDS, …
do_attack(buffer);
// Access index 'secret' in buffer

for (i = 0; i < 256; i++) {
     dt = time_access(buffer + 1024*i);
     if (dt < THRESHOLD) {
         // Hit for byte 'i'
     }
}
```

Spectre logo used under CC0 terms (https://meltdownattack.com/)

Compartmentalization

In the Linux kernel

- Existing endeavors:
  - KPTI -> ASI
  - Kernel lockdown
  - Confidential Compute (SGX -> TDX/ SEV)
  - ….
- Heavyweight solutions typically reserved for virtualization
- **Fundamental problem: <u>context-switching is expensive</u>**



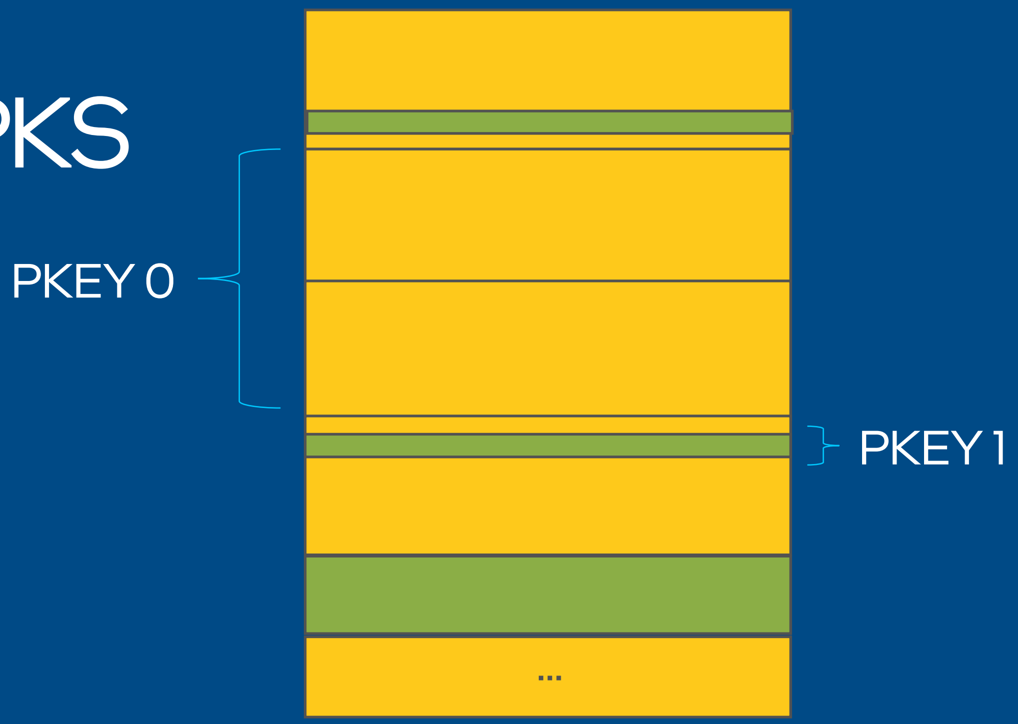
Leverage new hardware features to avoid context-switching

Legend
SGX: Intel® Software Guard Extensions
TDX: Intel® Trust Domain Extensions
SEV: AMD Secure Encrypted Virtualization

Memory Protection Keys: PKU/ **PKS**

PKS: Protection Keys for Supervisor

- Allows PTE permissions to be overridden on a domain key basis
- **No need to invalidate TLB/** change CR for a quick permission change
- Has been available for user-space (PKU) for years, for supervisor (PKS) since 4th Generation Intel® Xeon® Scalable



PKS in-depth

Protection Keys for Supervisor

- Protection key in each PTE
- Per-thread MSR to disable R/W permissions of each key (max 16 keys)
- For in-depth info see Ira Weiny & Rick Edgecombe Linux Plumbers Conference talk from 2021 “Protection Keys, Supervisor (PKS)”

PTE layout

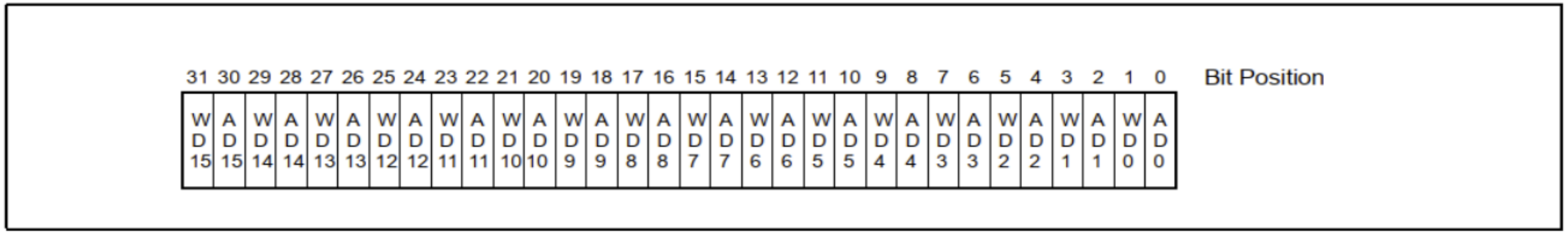


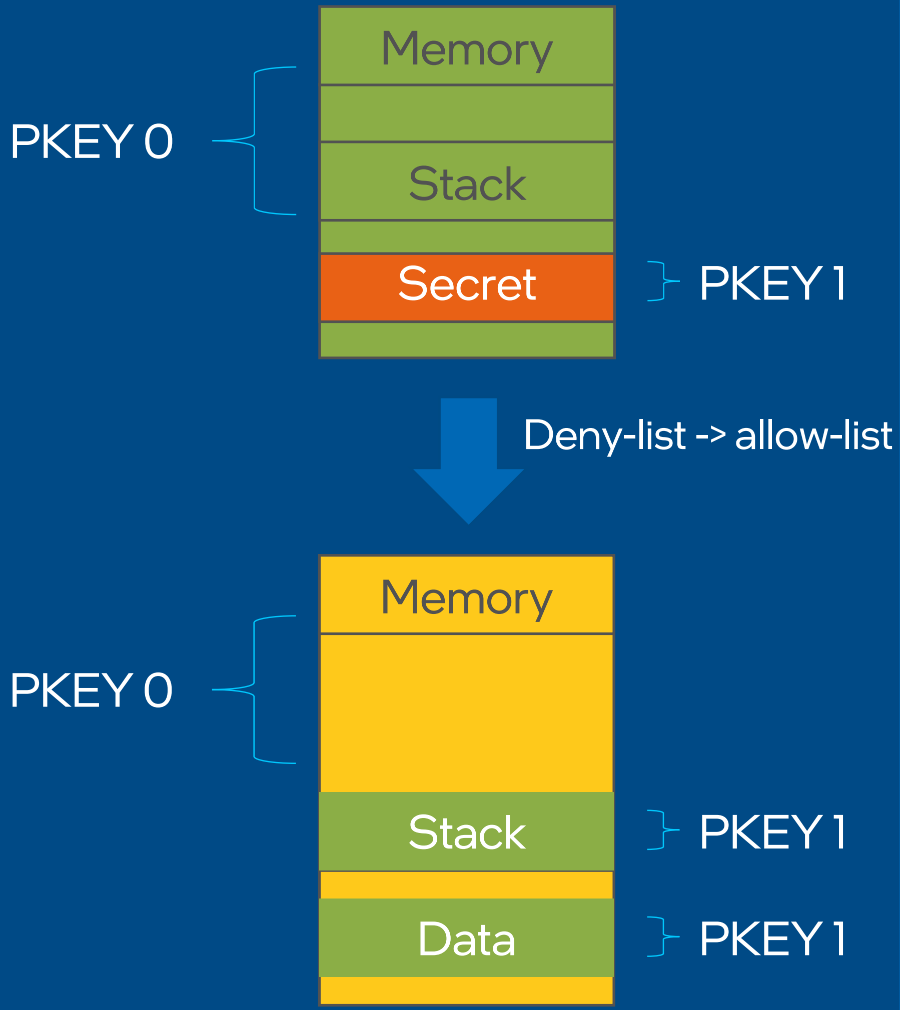
Figure 2-9. Format of Protection-Key Rights Registers

IA32_PKRS MSR

PKS kernel compartmentalization

Possible Targets

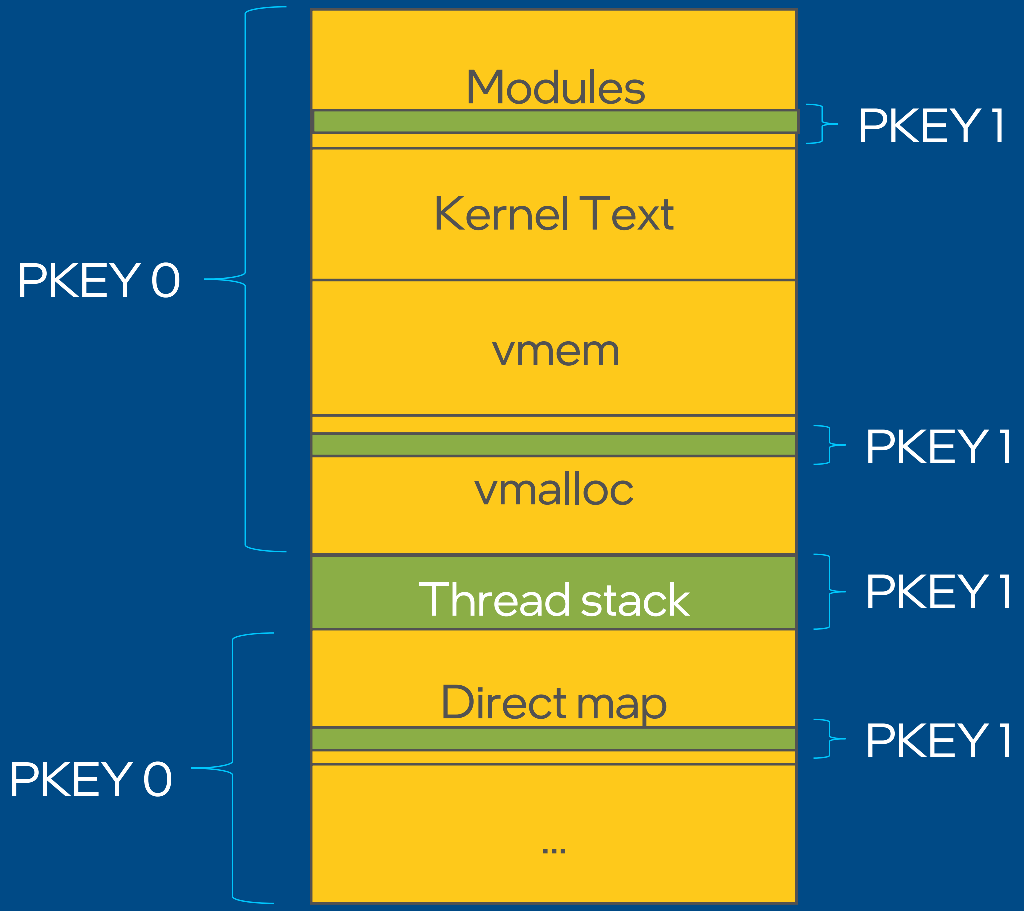
- PKS has been used to protect sensitive areas (PMEM, PTs, crypto keys) from the kernel
- Turn this around: protect kernel from potentially malicious code
  - eBPF: usually no need to access all mapped kernel memory
  - ASI: drop-in replacement
  - IOuring? There have been some recent vulns
  - More? Happy to get input



PKS kernel compartmentalization

How it works

- Two domains: privileged [key 0] and non-privileged [key 1]
- Privileged kernel domain: **PKEY 0**
  - Default kernel operation IA32_PKRS MSR.0[WD/AD] := 0 -> No W/R disable
- Switching to non-privileged domain:
  - IA32_PKRS MSR.0[WD/AD] := 1
  - Make sure that required memory is available for **PKEY 1** (e.g., stack)



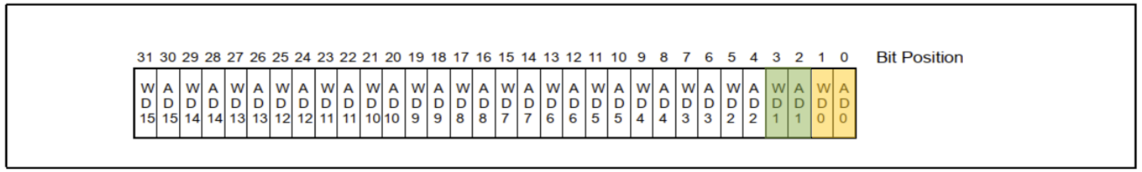
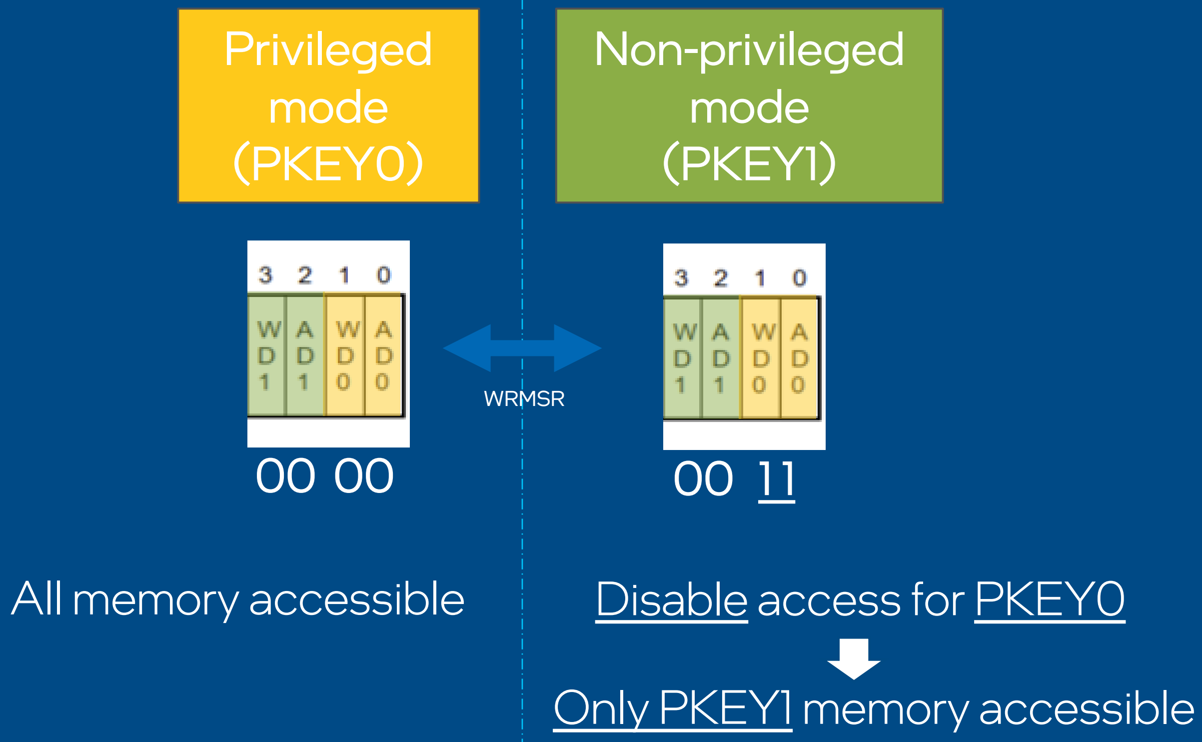
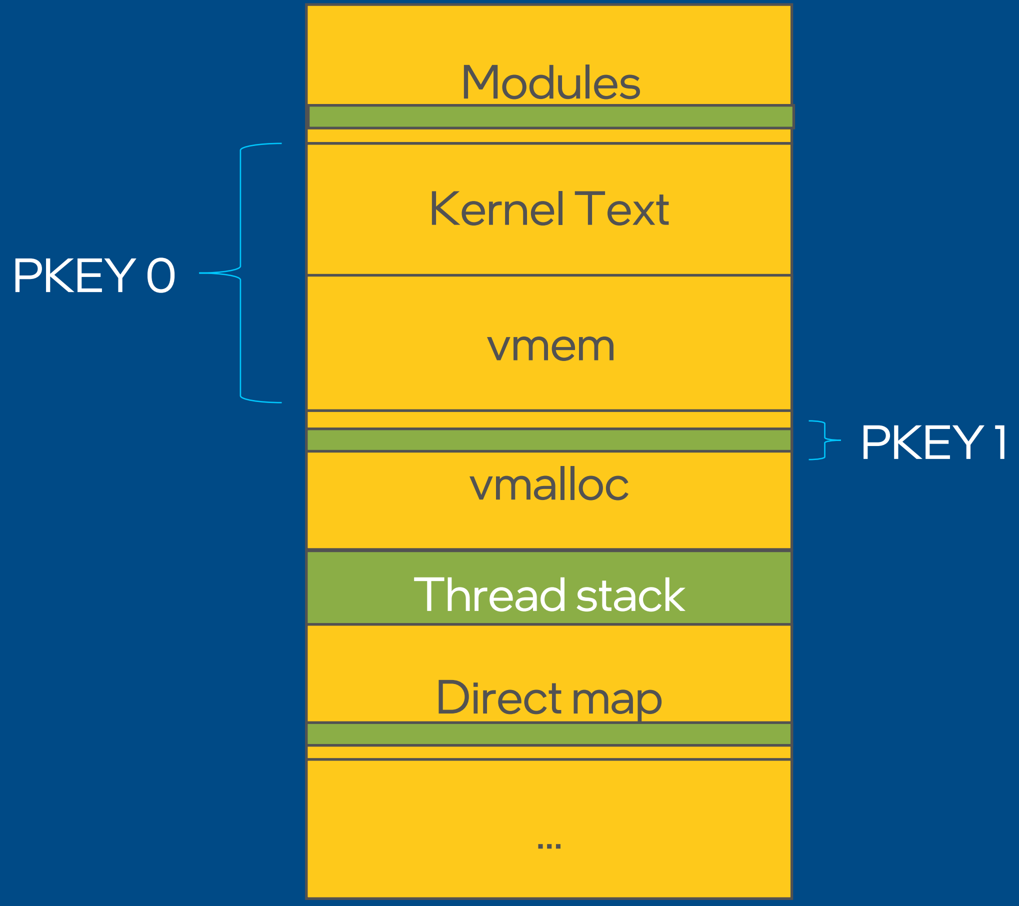


Figure 2-9. Format of Protection-Key Rights Registers





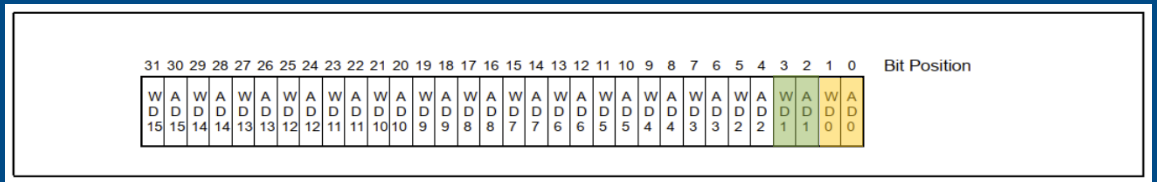
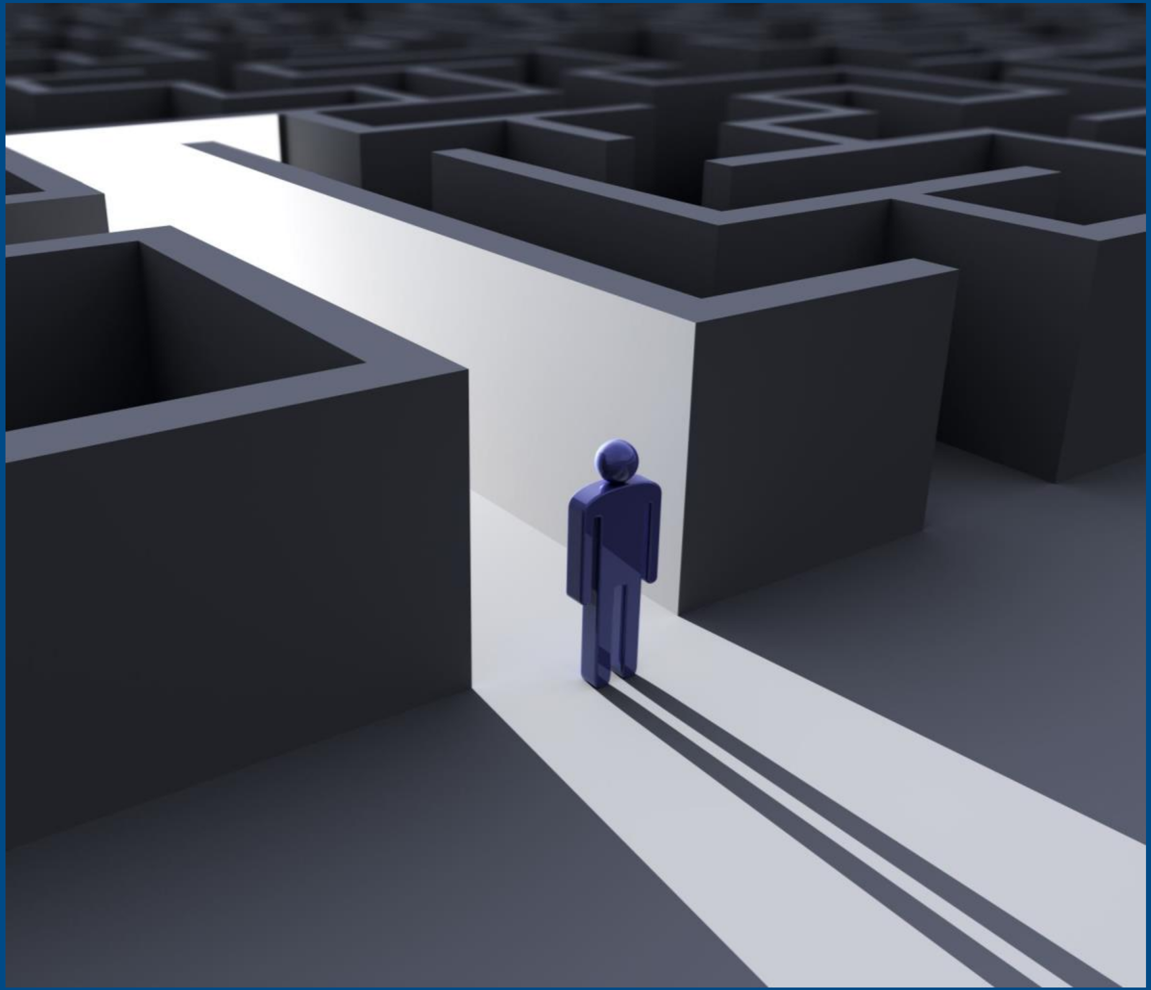


Figure 2-9. Format of Protection-Key Rights Registers

Challenges

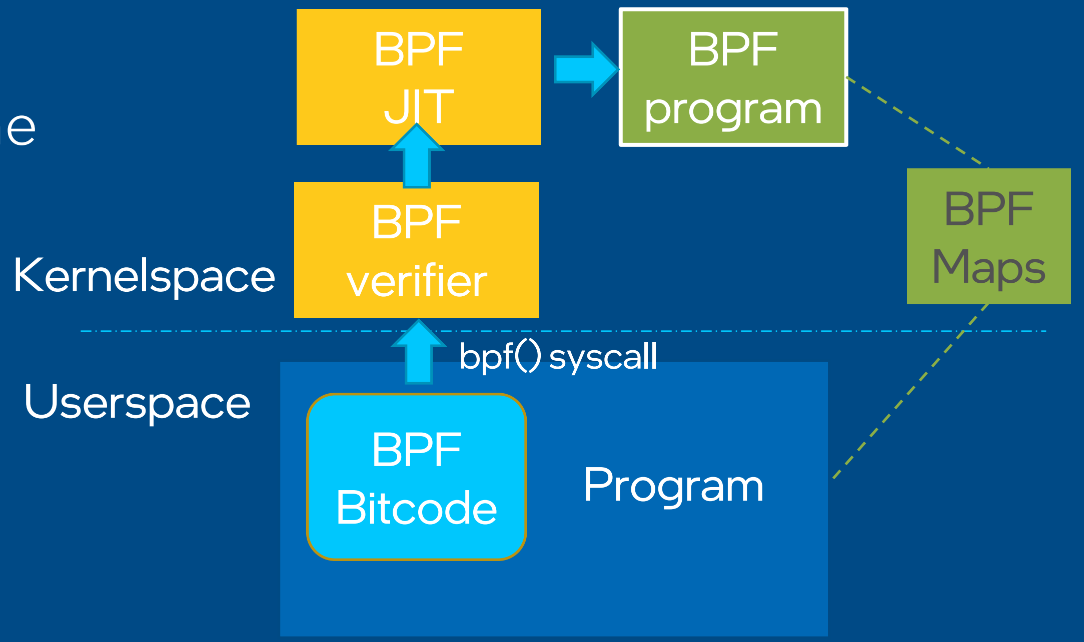
- In the general sense for compartmentalization:
- Memory accesses are not localized
  - Solution: temporarily allow more permissible accesses: similar to SMAP
- How to determine the memory areas that need to be accessed by a domain?
  - Easy for memory allocated in module itself
  - What about memory objects allocated outside the module?
- Let’s start with some already “sandboxed” use-cases



PKS eBPF isolation

eBPF

- eBPF: virtual machine running user code in the kernel for network filters
- Restrictive environment, in-kernel verifier
- Several bugs in verifier:
  - Quick MITRE search: 80 BPF-related CVEs
  - CVE-2021-31440: OOB access
- Transient Execution Attacks
  - Unprivileged eBPF disabled by default
  - Mitigations in-place -> performance overhead
  - Can we perhaps get rid of mitigations?
- Small, self-contained VM: **easy target for isolation**
- Can run in privileged (tracing) & unprivileged mode (requires mitigations)



eBPF mitigations

Speculation

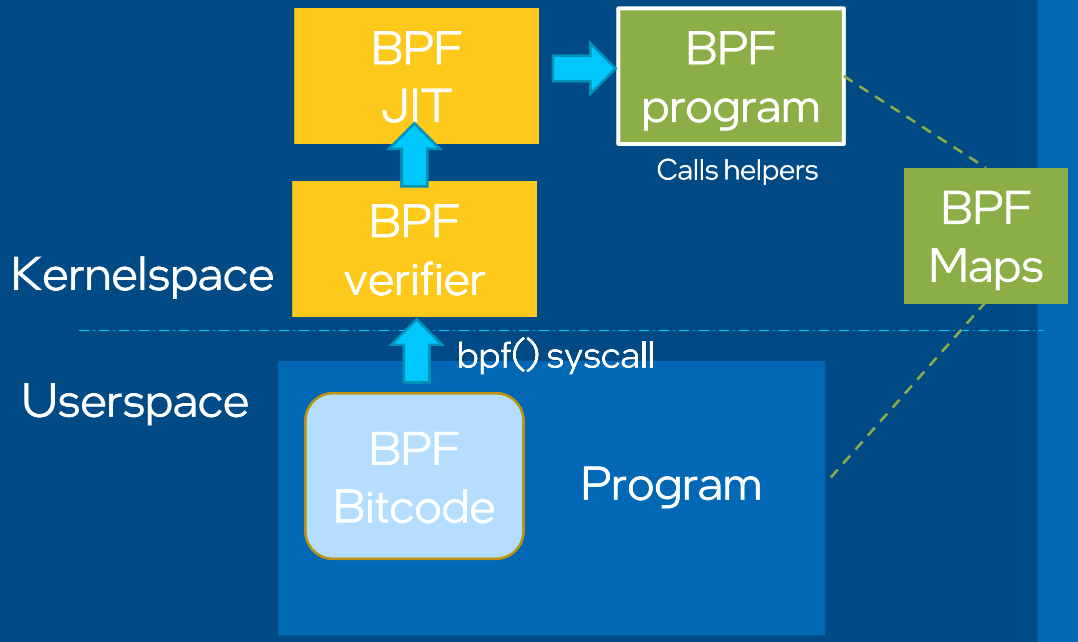
- Spectre v1 (speculative bounds check):
  - Array masking
  - Verify "impossible" (speculative) paths
- Spectre v2 (branch target injection):
  - Indirect calls in bpf helpers, bpf_tail_call
  - Retpoline. Can often be avoided, but retbleed?
- Spectre v4 (speculative store bypass):
  - lfence
- Spectre BHB
  - Disable unprivileged BPF
- See PriSC '22: "BPF and Spectre: Mitigating transient execution attacks"
- -> Significant overhead

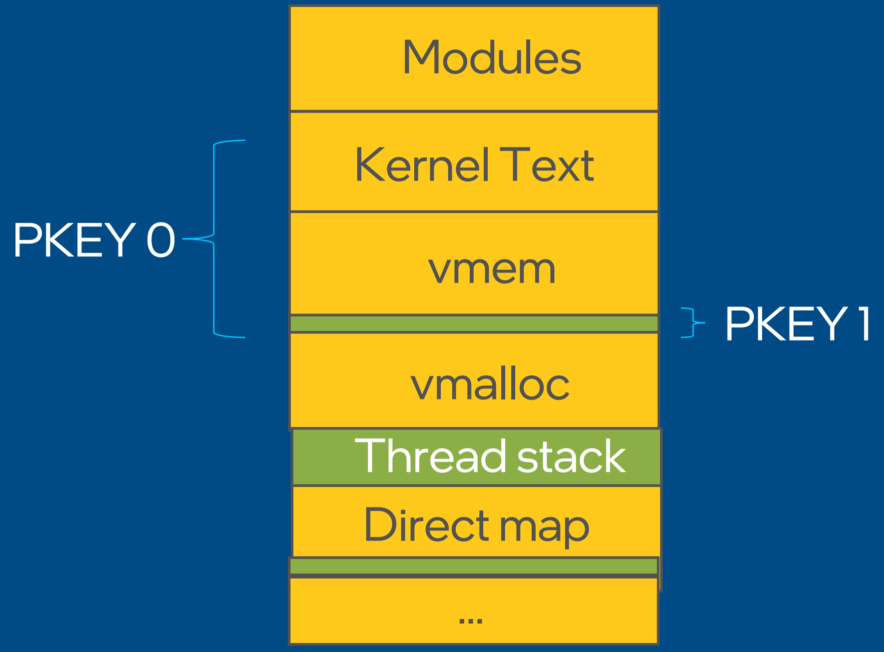
```
// x is user-controlled
1: if (x < array1_size)
2:   y = array1[x]
3:
// Train mis-speculation on 1
// -> spec. access [array1 + x] OOB
```

```
// r0 = pointer to a map array entry
// r7 = pointer to a map array entry
r4 = r10
r4 += -1
*(u8 *)(r4 -511) = 0
r2 = *(u64 *)(r7 +8)
r3 = *(u64 *)(r0 +4608)
r3 &= 1
r3 &= 2
r3 -= 511
if r2 != r3 goto pc+7
r4 += r2
r4 = *(u8 *)(r4 +0)
// leak r4
```

PKS eBPF isolation

- Switch domain on
  - bpf_trampoline_enter
    bpf_trampoline_exit
- Helpers:
  - Access stuff like `current`
  - Two approaches:
    - Map all accessed pages as PKEY1
    - Dynamically disable protection
- Maps, perf-buffers, per-task storage
- Tracing BPF programs can read arbitrary data:
  - `bpf_probe_read()`, `bpf_probe_read_string()`





Register values (memory that should be accessible)

- PTR_TO_CTX
  - Pointer to bpf_context.
- CONST_PTR_TO_MAP
  - Pointer to struct bpf_map. “Const” because arithmetic on these pointers is forbidden.
- PTR_TO_MAP_VALUE
  - Pointer to the value stored in a map element.
- PTR_TO_MAP_VALUE_OR_NULL
  - Either a pointer to a map value, or NULL; map accesses (see BPF maps) return this type, which becomes a PTR_TO_MAP_VALUE when checked != NULL. Arithmetic on these pointers is forbidden.
- PTR_TO_STACK
  - Frame pointer.
- PTR_TO_PACKET
  - skb->data.
- PTR_TO_PACKET_END
  - skb->data + headlen; arithmetic forbidden.
- PTR_TO_SOCKET
  - Pointer to struct bpf_sock_ops, implicitly refcounted.
- PTR_TO_SOCKET_OR_NULL
  - Either a pointer to a socket, or NULL; socket lookup returns this type, which becomes a PTR_TO_SOCKET when checked != NULL. PTR_TO_SOCKET is reference-counted, so programs must release the reference through the socket release function before the end of the program. Arithmetic on these pointers is forbidden.

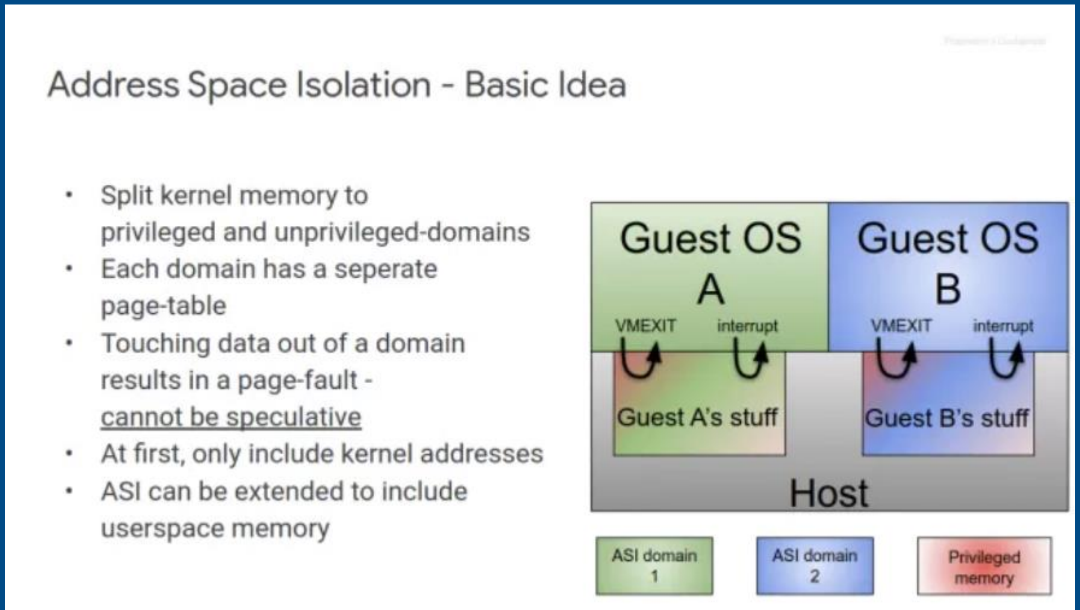
Some observations

- Overhead: WRMSR when switching into BPF
  - Low: MSR write takes just a handful of cycles
- Some initial benchmarking:
  - WRMSR overhead for BPF syscall tracing ~1% on LMBench
- Disable BPF mitigations -> possible speedup

PKS kernel compartmentalization

Why use PKS?

- Compatible with existing ASI design
- Lightweight: switch between domains by writing an MSR
- Flexible: use it only where it makes sense

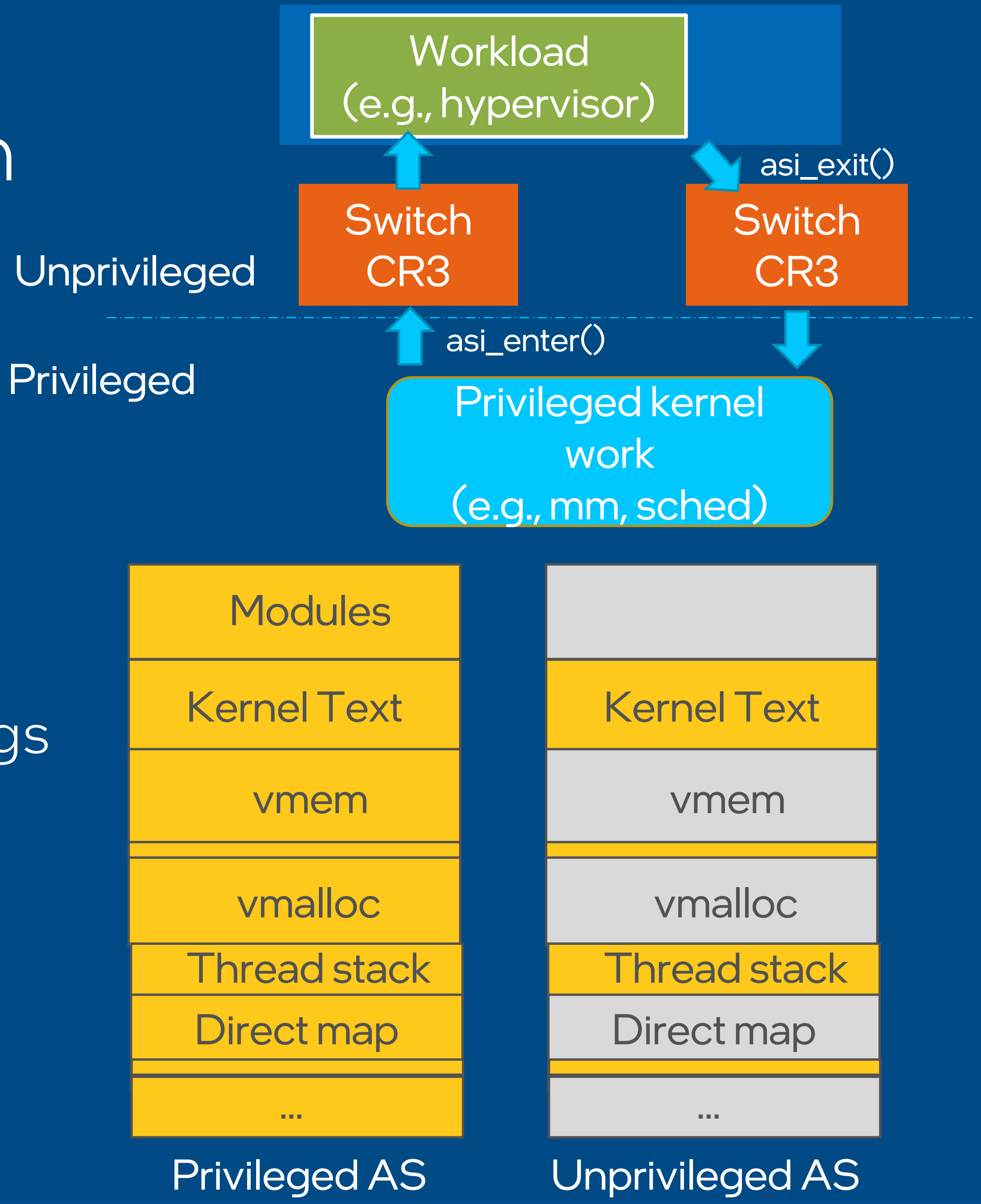


Google's ASI - Linux Plumbers Conference Dublin 2022

ASI: Address Space Isolation

In a nutshell

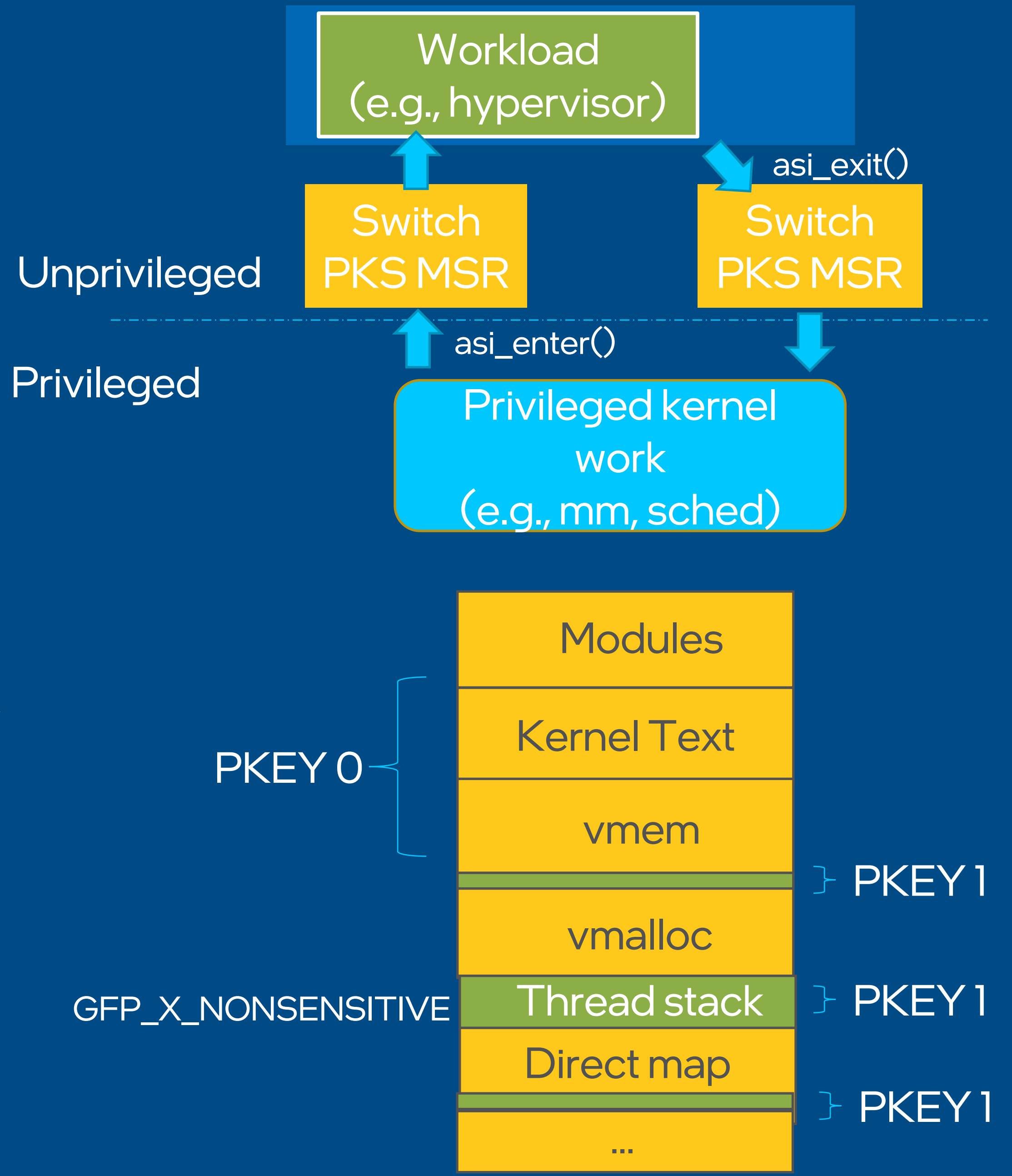
- Split kernel memory into privileged and unprivileged domains
  - Two page-tables
  - Privileged: “normal” kernel mappings
  - Unprivileged: minimal set required
- asi_enter()/ asi_exit()
- Kmalloc/vmalloc GFP_X_NONSENSITIVE flag



ASI-PKS

Address Space Isolation with PKS

- Same as ASI, but use much more lightweight MSR switch
- No need for two sets of page tables
  - Sets the domain key in the PTE entry
- Modify kmalloc/ vmalloc to use ASI-PKS pages/ slabs
- Downside: PKS cannot override execute permission -> different security guarantees



Conclusion

PKS Compartmentalization

- Third-party untrusted kernel code -> memory errors, transient execution gadgets allow memory disclosure
- Use PKS to make kernel data non-accessible
- First use-case: eBPF isolation, ASI drop-in replacement
- Lower overhead than switching CR3
- Possibly get rid of other mitigations?

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Stock images used in this presentation are used under license by Microsoft

Contact

- Twitter: @sirmc
- sebastian.osterlund@intel.com

intel®