KVM Forum 2022

Now you see me, now you don’t

Splitting pKVM into discrete, mutually exclusive address spaces

Marc Zyngier
<maz@kernel.org>

Foreword

- pKVM aims at providing strong isolation between host and guest
  - Please refer to Quentin Perret's "**Protected KVM on Arm64: A Technical Deep Dive**"
- This presentation is heavy on the arm64 architecture side
  - That's what pKVM runs on at the moment
  - Please ask questions if something seems obscure (it probably is)
- A lot of this is still a work in progress!

Idle considerations

Data leakage: a few axioms

The sorry state of current software

- Read gadgets exist in hypervisors
- The hypervisor has access to state that should be kept private
- The more code you add to the privileged part of the hypervisor, the more fragile it becomes

Data leakage: a few axioms

What state can we leak

- The hypervisor maps a number of guest-specific data structures
  - Among which the vcpu and kvm structures
- Some of the state gets pushed onto the hypervisor's stack (runtime)
- In limited cases, the hypervisor itself could map some guest memory

Example: the ARM TRNG hypercall

- Allows a guest to obtain some top-notch quality entropy
  - The entropy can get written to the hypervisor stack (register spills from the compiler)
- Could be retrieved via a read gadget -- and the right timing
  - Yes, this is obviously concerning
- Fear not, the architecture has a solution for us…

Recommendation from ARM (DEN 0098, 1.2.1)*

The TRNG FW implementation should:
Discard all entropy bits from all memory locations once the entropy has been delivered to the requester or to the next EL in the entropy delivery path. The entropy bits can be discarded by, for example, overwriting them with zero.

* https://developer.arm.com/documentation/den0098/latest

Oh No!

This doesn’t really work...

- The hypervisor keeps a record of the vcpu state
  - including copies of the registers
  - cannot just discard the entropy state (counter-productive)
- The hypervisor is written in a high-level language
  - doesn't necessarily track all the locations where things are written
  - spilling on the stack really isn't visible

Oh No!

This doesn't really work...

- Without further Cache Maintenance Operations (CMOs), it could be possible to retrieve the "erased" data
  - and CMOs to what cache level?
  - CMOs don't really work as a security feature
  - We could map everything as non-cacheable (not a real suggestion...)
- We need something else...

We need a way to isolate the memory that is used by the hypervisor while dealing with a vcpu

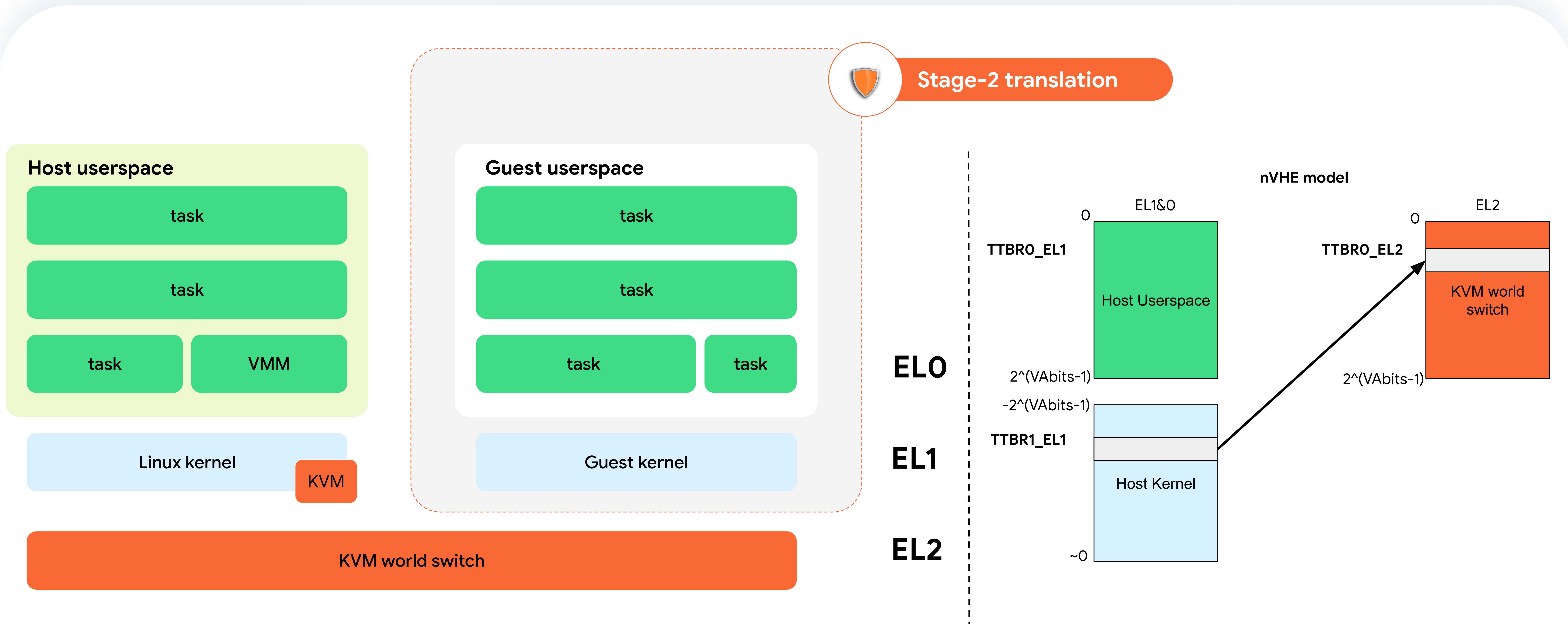
(did I hear someone say page tables?)

Refresher

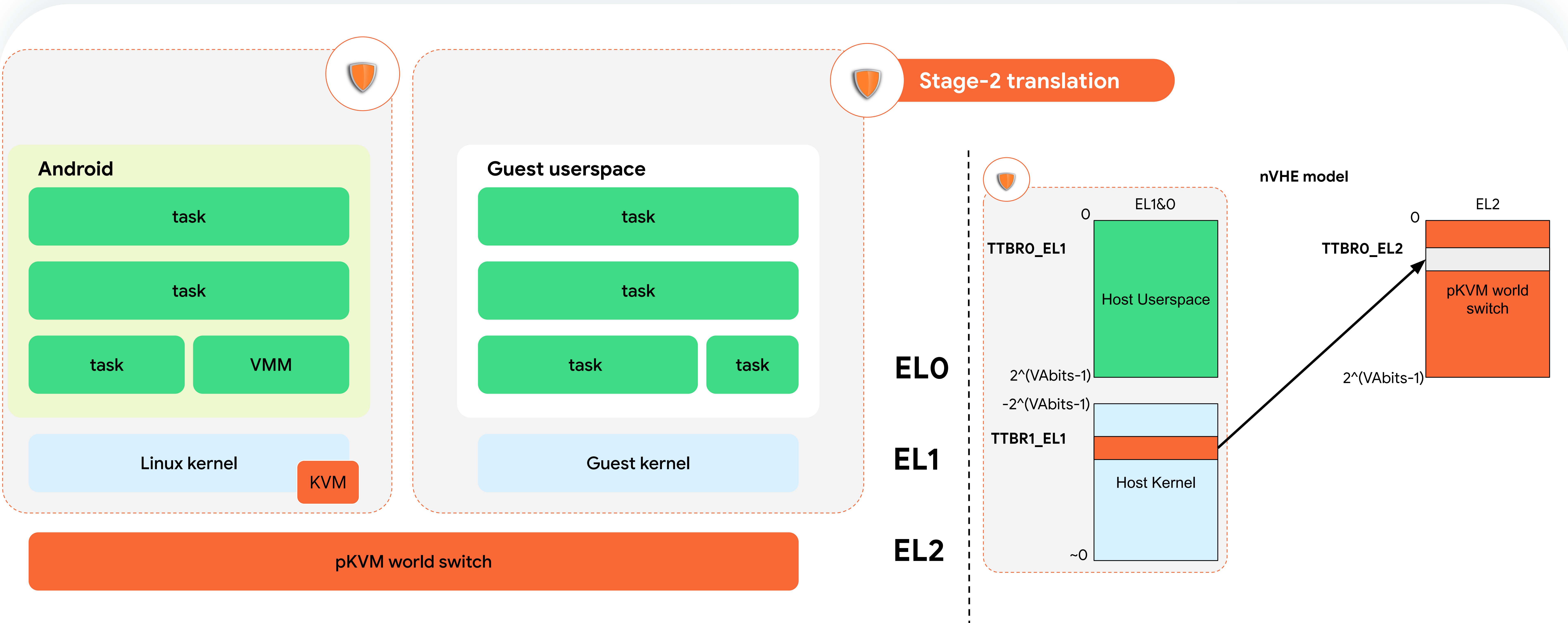
1001 ways to (ab)use exception levels...

The ARMv8.0 exception model

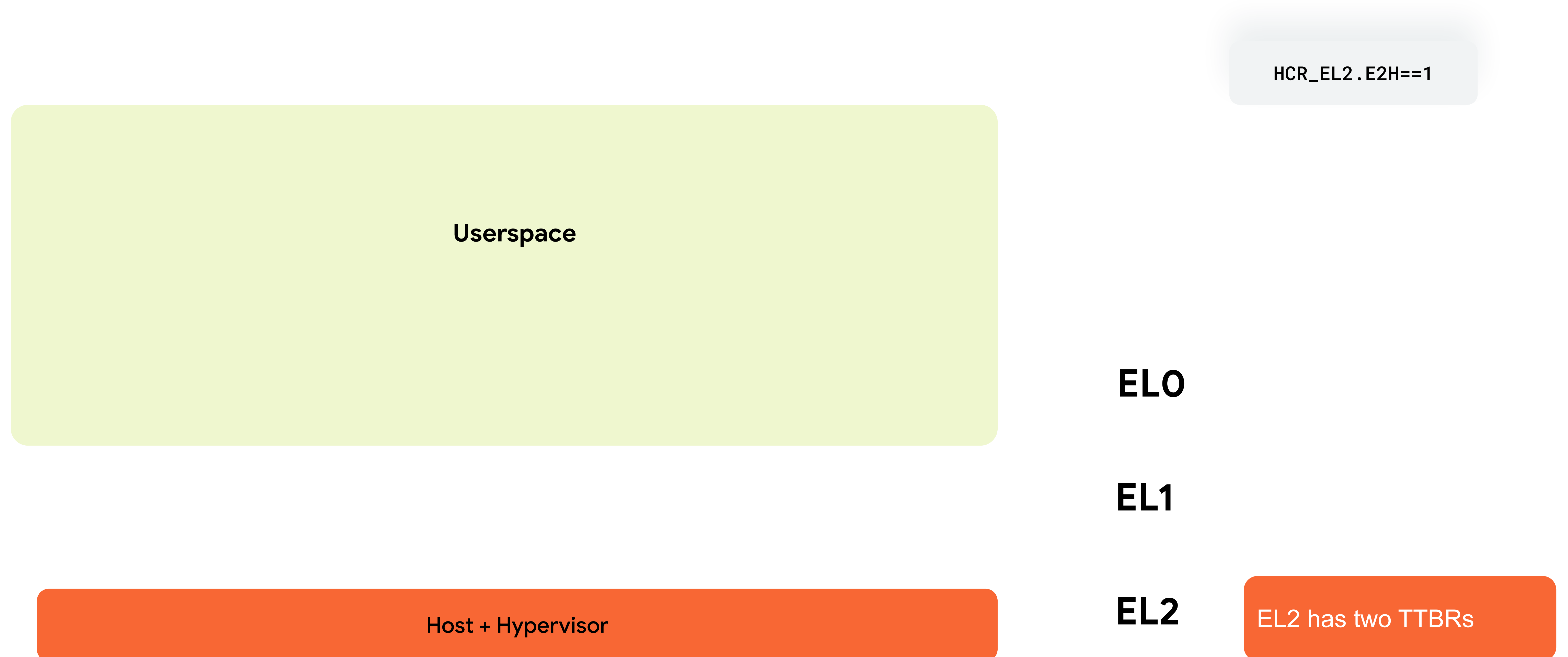
How KVM uses ARMv8.0 (aka nVHE)



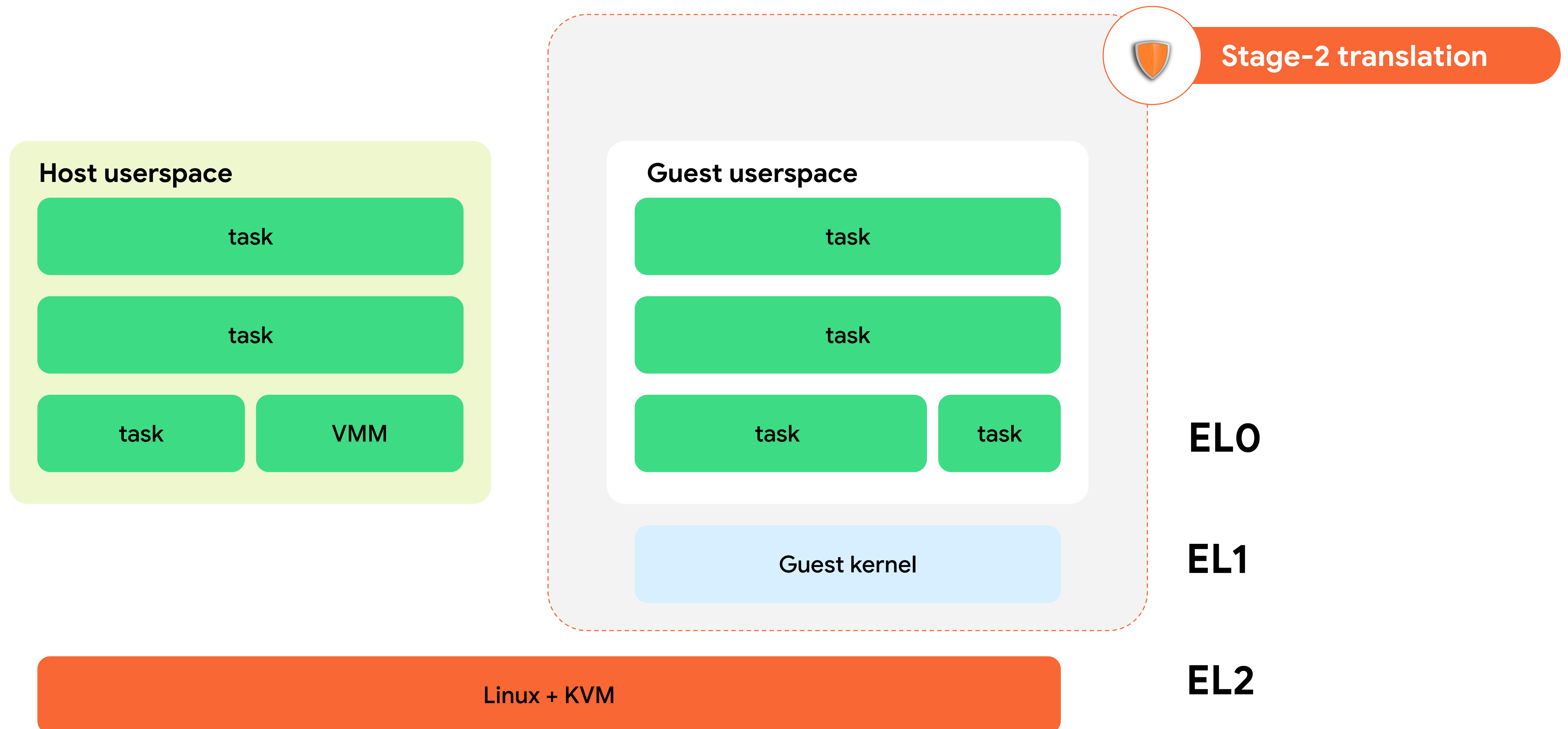
How pKVM uses ARMv8.0 (aka nVHE)



The ARMv8.1+ host exception model



How KVM uses ARMv8.1+ (aka VHE)



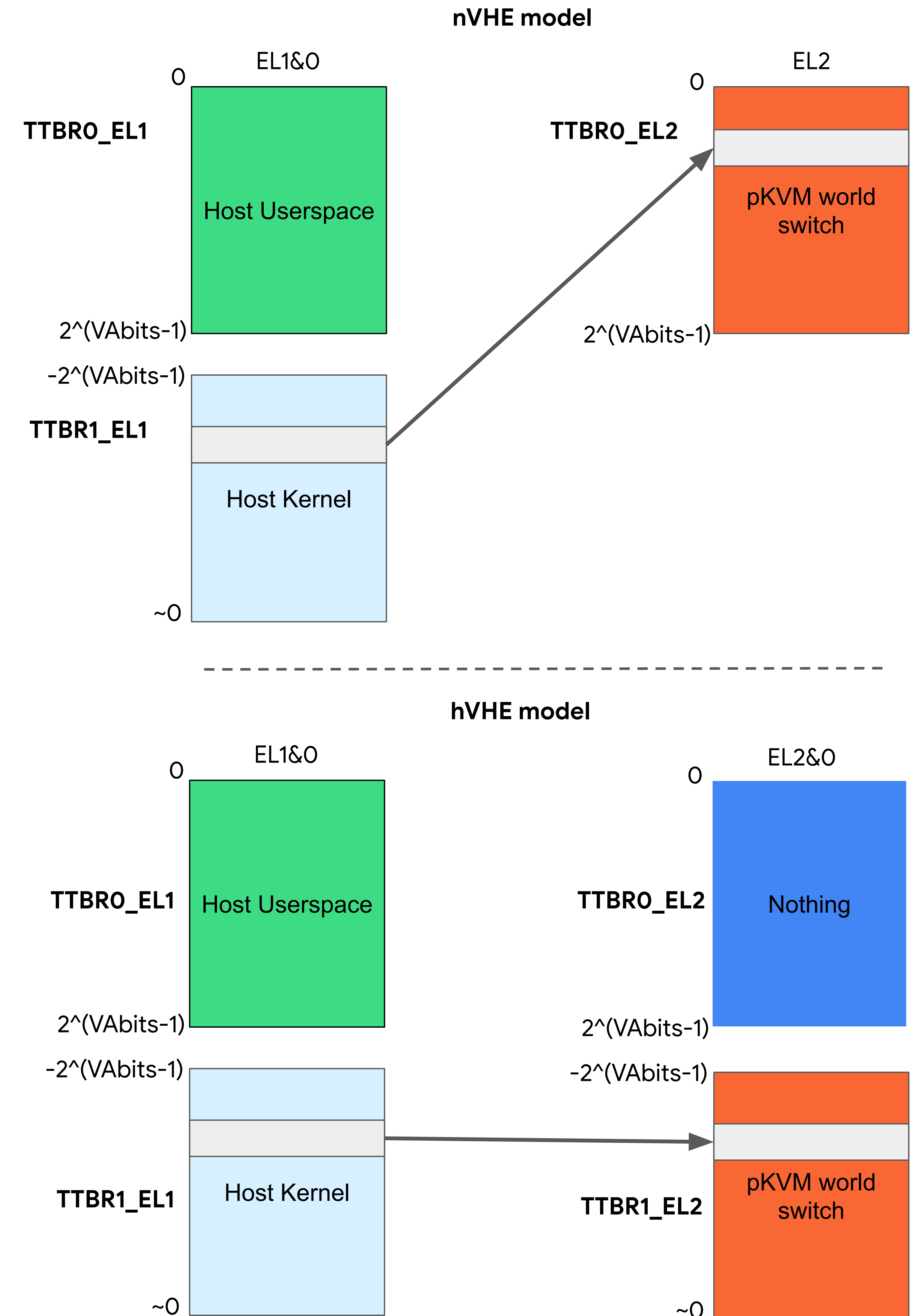
Playing the letter soup game

nVHE + VHE → hVHE

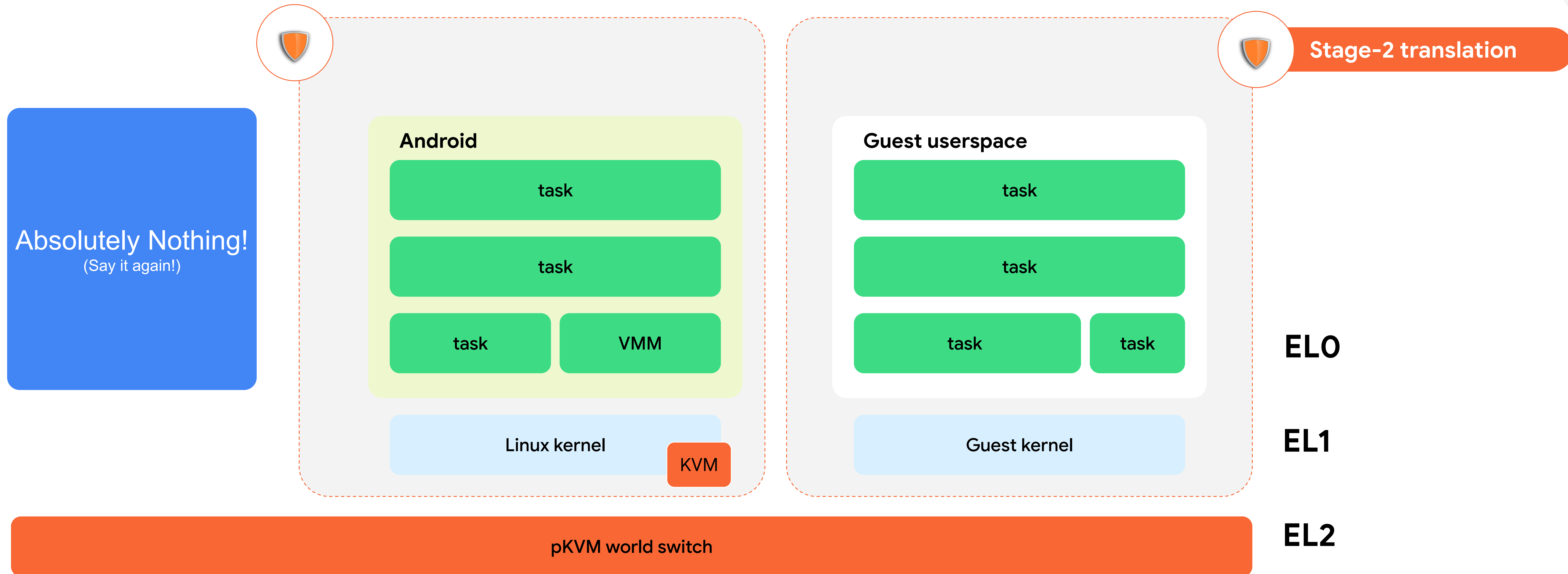
- But why forcing pKVM to use nVHE on ARMv8.1+ hardware?
- We could enable VHE *for EL2 only*, and still run the host kernel at EL1
  - This is the nVHE logical model using the VHE infrastructure
- Just pretend we're running a guest while running the host
  - This is already the case with pKVM
    - Keep `HCR_EL2.TGE==0` at all times
    - TGE = Trap General Exceptions – being 0 indicates we're running a guest

nVHE + VHE → hVHE

- Two VA ranges
  - `TTBR0:[0 .. 2^(VAbits-1)]`
  - `TTBR1:[-2^(VAbits-1) .. ~0]`
- Move the hypervisor mapping to **TTBR1_EL2**
  - Rids us of the pointer conversion dance
  - We reuse the same VAs as the kernel
  - **TTBR0_EL2** is currently unused…



How pKVM uses ARMv8.1+ with hVHE



What is it good for?

- Absolutely nothing! Yet.
  - Crucially, EL2 always has a single address space in use
- But we're having fun moving things around
- And pKVM now has a chance to run on fruity CPUs…
  - They may have "forgotten" half of the architecture by making `HCR_EL2.E2H RES1`

Interlude

Application Space IDentifier

- The ARM architecture specifies that TLBs can be tagged by ASID
  - Driven by a page-table bit (nG -> non-global)
  - Similar to PCID on x86
- Linux uses this to isolate userspace contexts
  - Each address space gets its own set of page tables
- Can be used as long as a translation regime has two TTBRs
  - Described as the `EL1&0` and `EL2&0` translation regimes in the ARM ARM

“Loading” a vcpu...

- When about to run a vcpu, KVM calls '`vcpu_load()`'
- Has the effect of making the vcpu notionally 'resident'
- Reversed by '`vcpu_put()`'

“Loading” a vcpu…

… on arm64

- Install on the physical CPU:
  - Any vcpu state that is compatible with the execution of the host
  - Everything else is loaded immediately prior to the execution of the vcpu
- And with pKVM:
  - Make all further hypercalls executed on this CPU act on that vcpu
    - you can't load vcpu0 and then run vcpu1 on that CPU
    - you can't load vcpu0 on two physical CPUs either

A space of one's own

What if each vcpu...

- Could only be made resident on a single CPU
  - pKVM already guarantees this (yay!)
- Had its own address space in the hypervisor
- Had its state exclusively mapped in this address space
- Came with a stack dedicated to execution at EL2
  - And of course only mapped in this address space?

EL2 under ASID?

Now that we have an extra VA range at EL2...

- We use ASIDs to isolate vcpu states
  - Make use of the ‘`0`’ part of the **`EL2&0`** translation regime (using **`TTBR0`**)
  - Each vcpu state gets its own ASID
  - Yes, this feels like a hypervisor userspace... more on that later
- Strong isolation primitive
  - per-vcpu TLBs
  - private to each physical CPU (no inter-CPU TLB sharing)

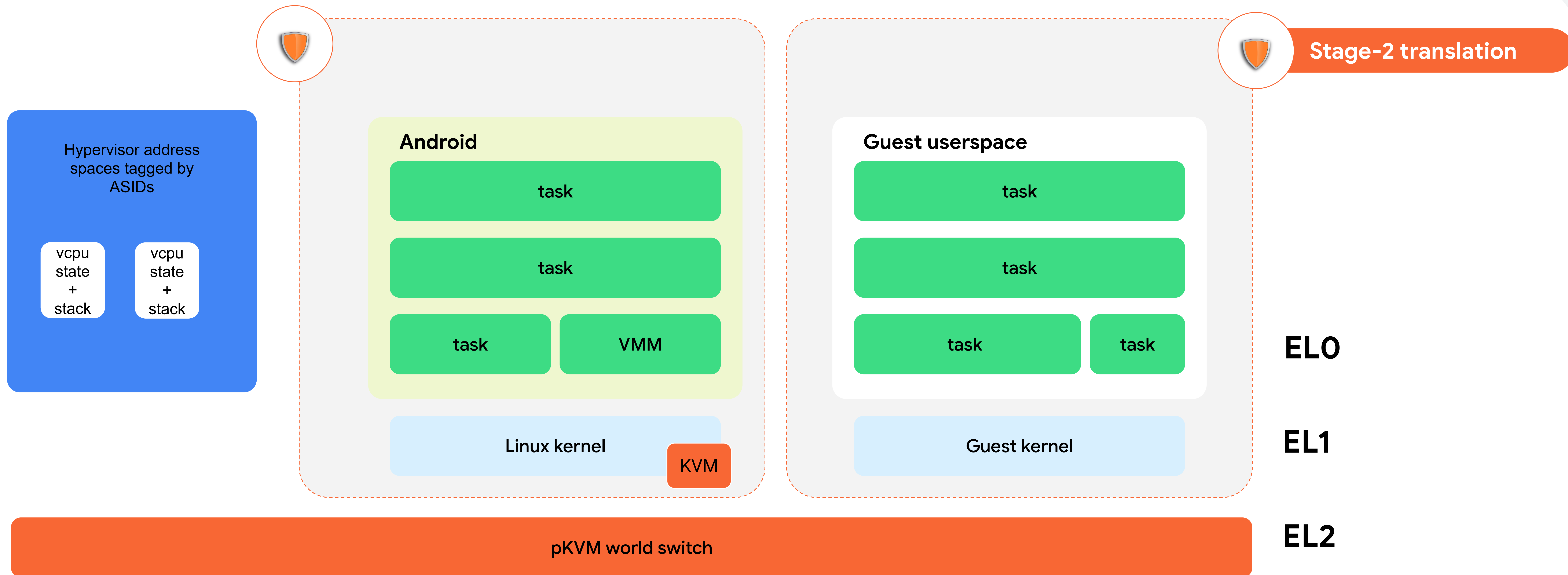
ASID for nothing...

... and fixmaps for free

Since each vcpu has its own VA space:

- Each vcpu state can live at a fixed address in that VA space
- No need to follow pointers, the address can be a constant
- Same thing for the per-vcpu EL2 stack!
- We end-up with a bunch of per-vcpu, per-CPU fixmaps
- Any register spill on the stack is now only visible to this CPU

How pKVM uses ARMv8.1+ with hVHE



Loading a vcpu, new and improved

```
int vcpu_load_ttbr(struct kvm_shadow_vm *vm,
                   int vcpu_idx)
{
    struct kvm_shadow_vcpu_state *shadow_state;
    u64 ttbr0;

    shadow_state = &vm->shadow_vcpu_states[vcpu_idx];
    ttbr0 = xchg(&shadow_state->ttbr0, 0);
    if (!ttbr0)
        return -EINVAL;

    /* Map vcpu */
    write_sysreg_el2(ttbr0, SYS_TTBR0);
    isb();
    __this_cpu_write(loaded_shadow_state, shadow_state);
    switch_to_private_stack();
}
```

This atomically sets both the ASID and the page-table root

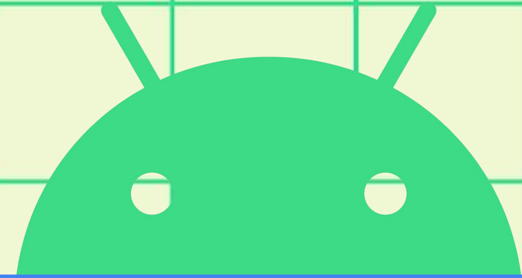
```
int vcpu_put_ttbr(void)
{
    struct kvm_shadow_vcpu_state *shadow_state;
    u64 ttbr0;

    shadow_state = __this_cpu_read(loaded_shadow_state);
    ttbr0 = read_sysreg_el2(SYS_TTBR0);
    switch_to_global_stack();
    /* Unmap vcpu */
    write_sysreg_el2(reserved_pgd, SYS_TTBR0);
    isb();

    __this_cpu_write(loaded_shadow_state, NULL);
    shadow_state->ttbr0 = ttbr0;
}
```

The cost of free address spaces

- Extra memory allocation
  - Page tables: 4 pages for 48bit VAs if using 4kB base granule) for each vcpu
  - One stack page for EL2 per vcpu
  - One zero page used when no vcpu is resident (shared)
- An ASID allocator
  - One ASID per vcpu, limiting the global number of vcpus to 2^16 (most of the time) or 2^8 (rarely)
  - One reserved ASID used when no vcpu is resident

The really ugly cost

- We have killed pKVM on ARMv8.0
- Is it acceptable? Hopefully...



Of sand and boxes

pKVM on mobile devices...

- pKVM isolates host and VMs from each other on the CPUs
- Crucially, it must also do the same thing for DMA
  - Requires IOMMUs to perform the isolation
  - These IOMMUs rarely comply with the ARM SMMU architecture
  - They have complex power management requirements
- This means having “minimal” drivers running in the hypervisor

Given the diversity of the ecosystem and the lack of standardisation, how do we:

- *enable pKVM on a large set of systems*
- *maintain some level of sanity in the hypervisor*

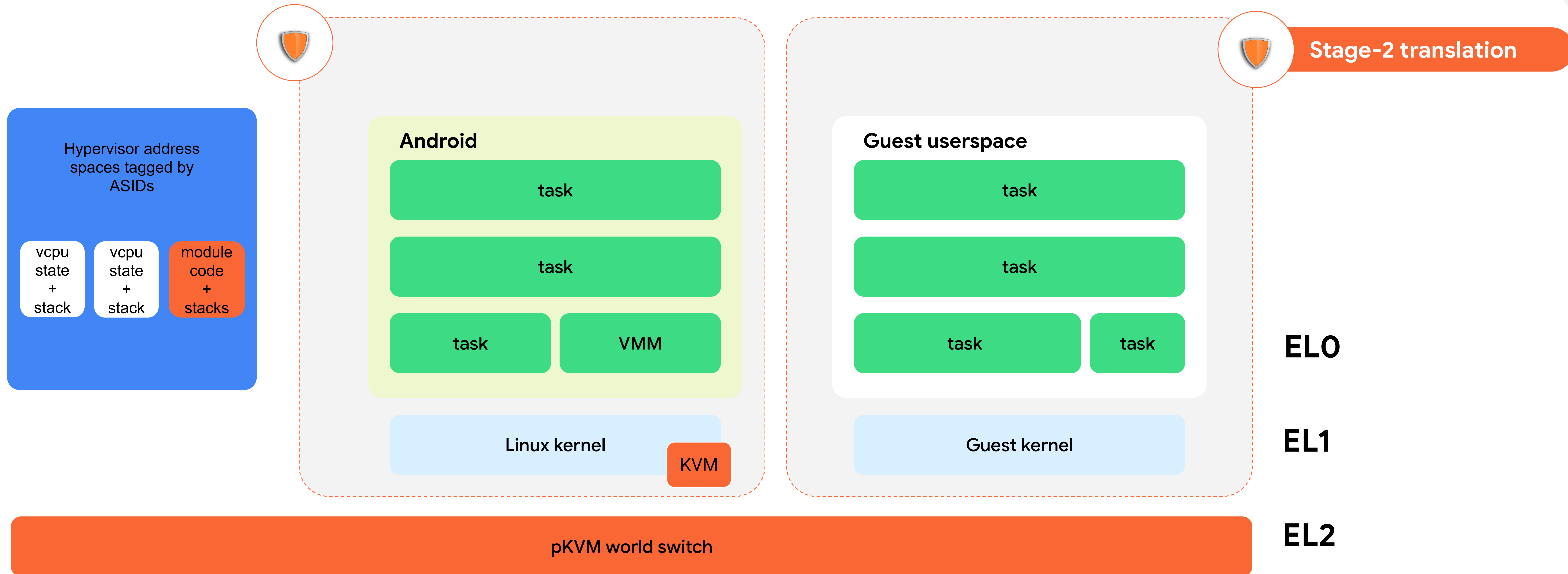
Hypervisor modules?

- Dealing with multiple drivers providing similar functionalities isn’t new
  - Kernel modules!
- However, bog-standard modules are not ideal for pKVM
  - No exported EL2 API
  - Trying to reign in the amount of privileged code
- It would be great if we could at least sandbox such modules

Sandboxing code in pKVM

- We just introduced hypervisor address spaces, only to map data
- We could also run code there if we made the mapping executable
  - However, this code would still be running at EL2
- What if we treated it as the hypervisor's “**userspace**”?
  - Set **`HCR_EL2.TGE==1`**, as we want to route all exceptions to EL2
  - **ERET** to EL0 (aka userspace), and start running our “module”!
  - **SVC** to come back to EL2 and reset **`HCR_EL2.TGE==0`**
  - **No access to vcpu state by design**, as they compete for a single TTBR

pKVM with hVHE and “modules”



Not quite a conclusion...

- We have defined the basic blocks to
  - Provide data and code isolation in the pKVM privilege part
  - Introduce some form of driver modularity to deal with a diverse ecosystem
- This was the easy part
- The hard part will be to define how we make use of this sandboxing
  - Defining a usable API is the next challenge
  - Yes, still a work in progress!

KVM Forum 2022

Thank you!

Marc Zyngier
<maz@kernel.org>