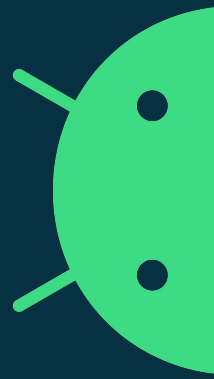
Protected KVM on arm64: A Technical Deep Dive

Quentin Perret <qperret@google.com>



android

Why talk about pKVM?

- It’s cool
- First birthday of the code!
- Might be inspiring
- Provide a “mental model” for [1]
- Resolve open discussion points

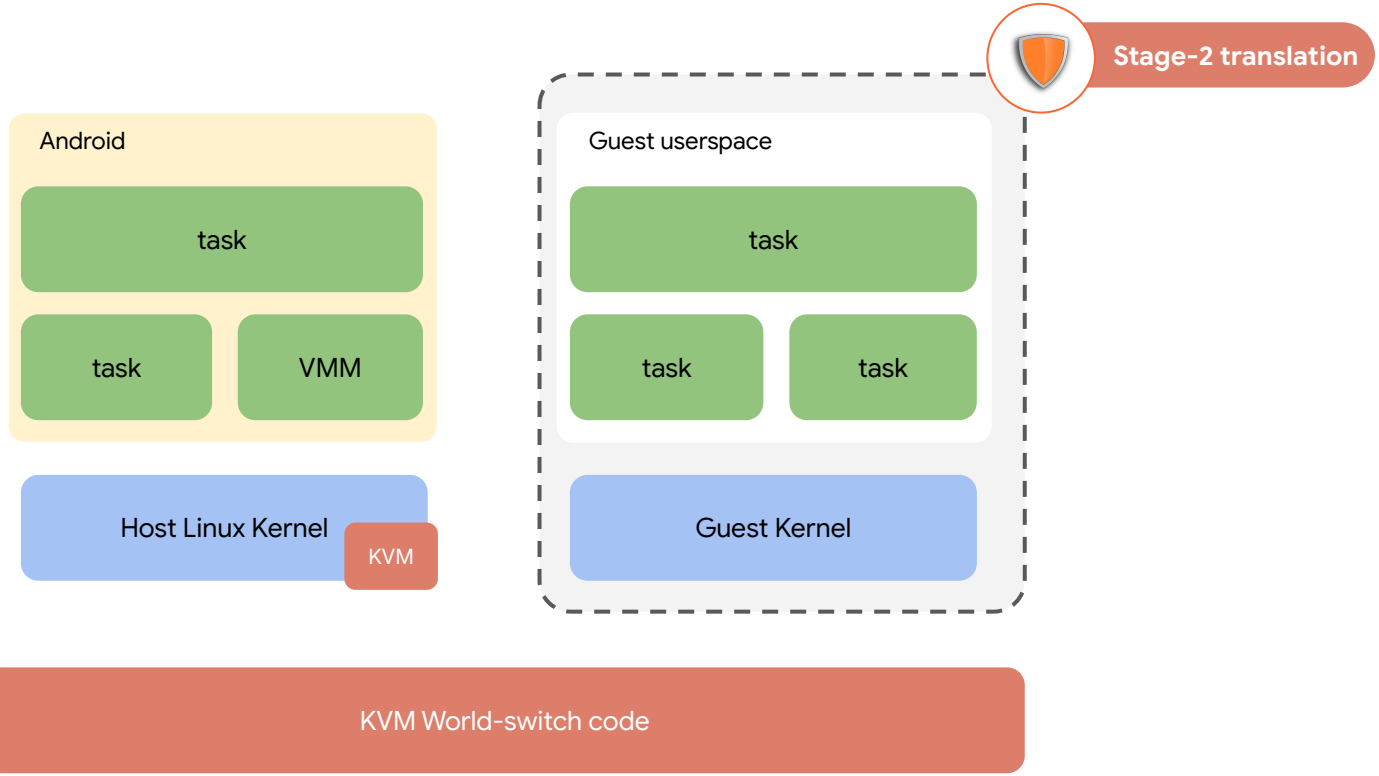
[1] https://lore.kernel.org/kvmarm/20220519134204.5379-1-will@kernel.org/

Disclaimer

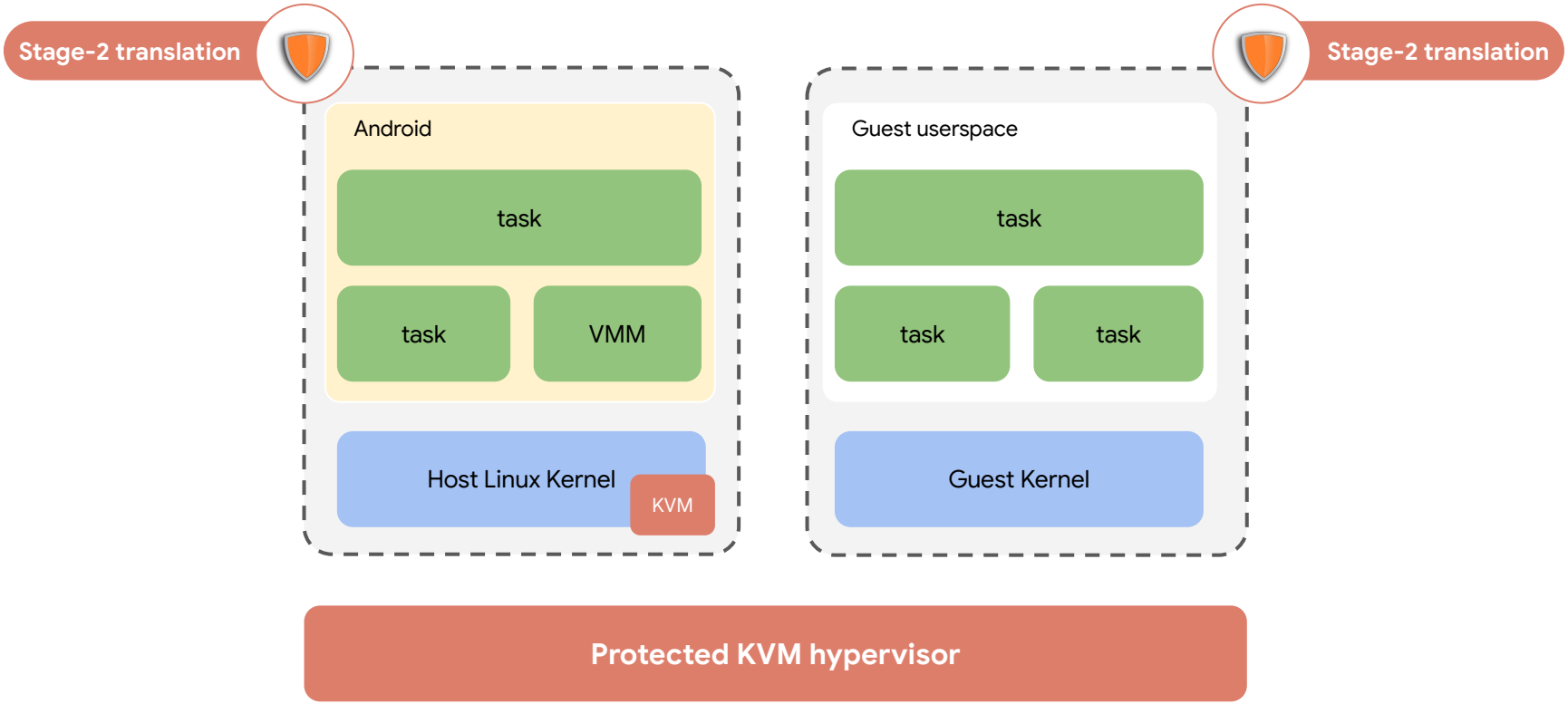
For more information about the “why” and Android details:

- *https://source.android.com/docs/core/virtualization?hl=en*
- *https://lwn.net/Articles/836693/*
- *https://www.youtube.com/watch?v=wY-u6n75iXc*
- *https://lpc.events/event/7/contributions/780/attachments/514/925/LPC2020_-_Protected_KVM_.pdf*

KVM port on armv8.0A (nVHE)

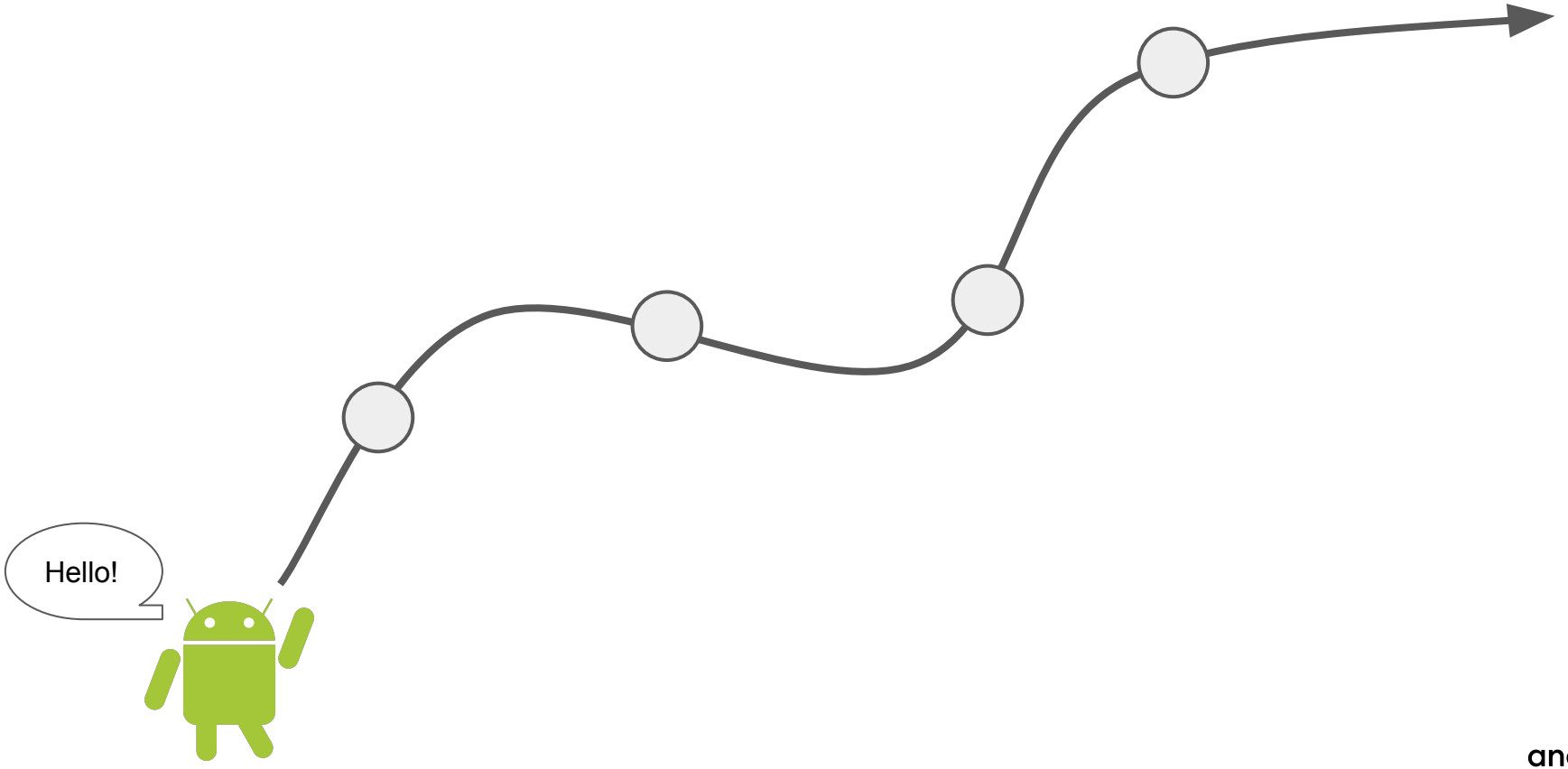


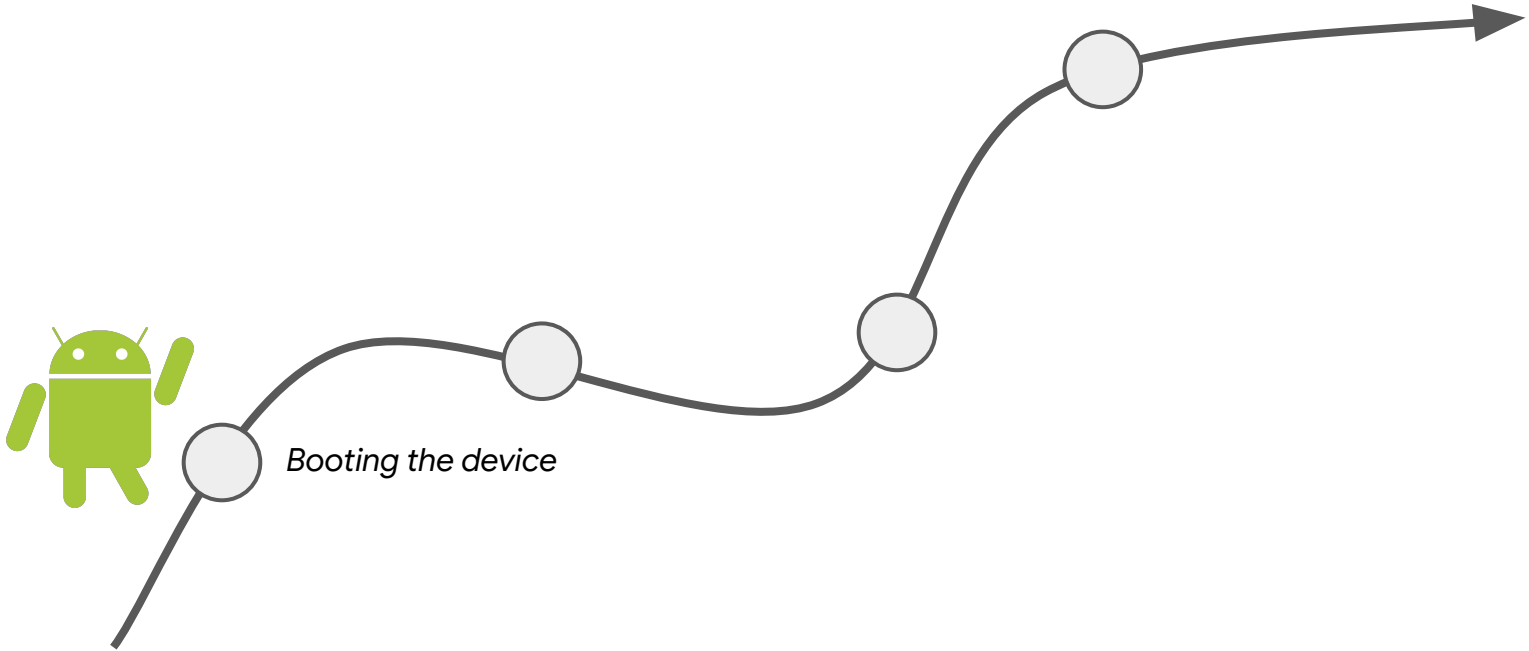
pKVM overview



Benefits

- Hypervisor and kernel are in the **same image** (code in `arch/arm64/kvm/hyp/nvhe/` **upstream**)
- Good for **hypervisor updatability**, leverages existing infrastructure for kernel updates
- Hypervisor and kernel updates are ‘atomic’, so **no ABI** between them!
- Hypervisor code is open source, auditable, patcheable, …


Hello!


Booting the device

Booting the device

- Bootloader verifies kernel image
- Kernel is entered at EL2
- Kernel install stub vectors, and erets to EL1

Early boot

Booting the device

- Bootloader verifies kernel image
- Kernel is entered at EL2
- Kernel install stub vectors, and erets to EL1

Early boot | Memory reservation

- EL2 memory pool reserved using memblock API
- Used for EL2-private data-structures, hypervisor stage-1 and host stage-2 page-tables

Booting the device

- Bootloader verifies kernel image
- Kernel is entered at EL2
- Kernel install stub vectors, and erets to EL1

- Host allocates temporary hypervisor stage-1 page-table
- Host allocates EL2 stacks, per-cpu pages, …
- `__pkvm_init()` hypercall

Early boot → Memory reservation → Preparing init

- EL2 memory pool reserved using memblock API
- Used for EL2-private data-structures, hypervisor stage-1 and host stage-2 page-tables

Booting the device

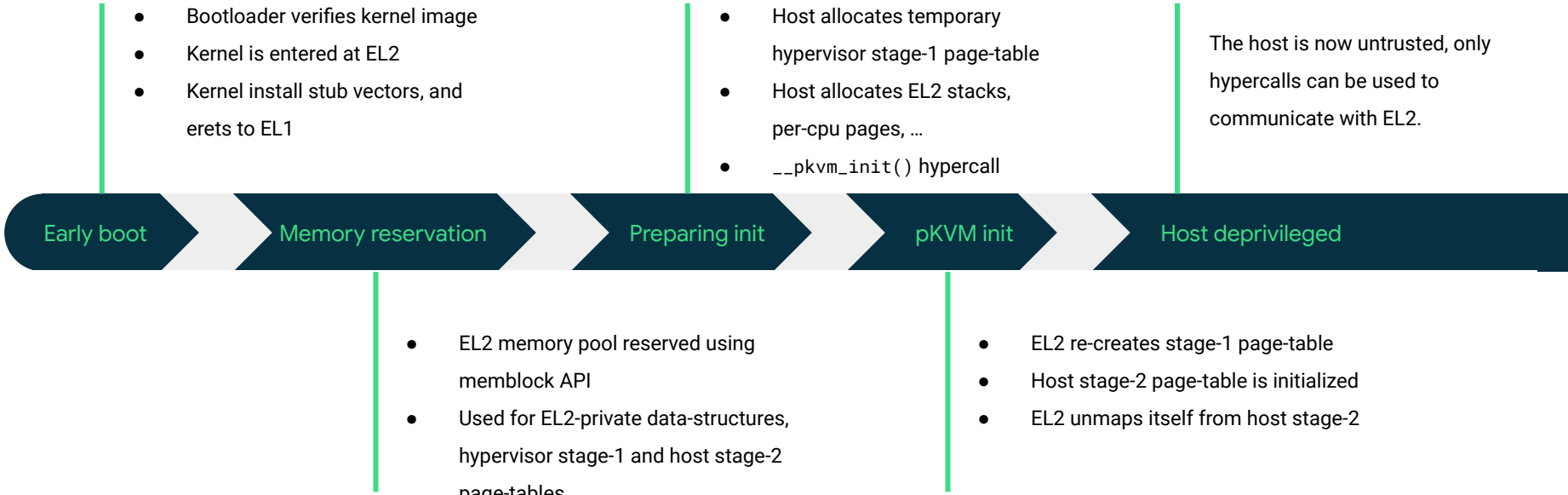
- Bootloader verifies kernel image
- Kernel is entered at EL2
- Kernel install stub vectors, and erets to EL1

- EL2 memory pool reserved using memblock API
- Used for EL2-private data-structures, hypervisor stage-1 and host stage-2 page-tables

- Host allocates temporary hypervisor stage-1 page-table
- Host allocates EL2 stacks, per-cpu pages, …
- `__pkvm_init()` hypercall

- EL2 re-creates stage-1 page-table
- Host stage-2 page-table is initialized
- EL2 unmaps itself from host stage-2

Booting the device



Booting the device

- Bootloader verifies kernel image
- Kernel is entered at EL2
- Kernel install stub vectors, and erets to EL1

- Host allocates temporary hypervisor stage-1 page-table
- Host allocates EL2 stacks, per-cpu pages, …
- `__pkvm_init()` hypercall

The host is now untrusted, only hypercalls can be used to communicate with EL2.

Early boot → Memory reservation → Preparing init → pKVM init → Host deprivileged

- EL2 memory pool reserved using memblock API
- Used for EL2-private data-structures, hypervisor stage-1 and host stage-2 page-tables

- EL2 re-creates stage-1 page-table
- Host stage-2 page-table is initialized
- EL2 unmaps itself from host stage-2

Page ownership tracking

- There are several possible ‘types’ of owners for memory
  - The host
  - The hypervisor
  - Guest VMs
  - …
- Pages are in one of four states for each possible owner:
  - **`PKVM_PAGE_OWNED`** — *exclusive access to the page*
  - **`PKVM_PAGE_SHARED_OWNED`** — *page shared by current owner with another entity*
  - **`PKVM_PAGE_SHARED_BORROWED`** — *page shared with current owner by another entity*
  - **`PKVM_NOPAGE`** — *no access to the page*

Page ownership tracking

- The state of pages is stored in the **Software Bits** of PTEs
- Sharing is only possible **between two entities** *(no n-way sharing yet)*
- In the host's stage-2, bits [63-1] in each **invalid PTE** is used to store the **identifier of the page owner**. The host's identifier is 0.

Page conversions

DONATION	*Initiator*	*Completer*
Before	`PKVM_PAGE_OWNED`	`PKVM_NOPAGE`
After	`PKVM_NOPAGE`	`PKVM_PAGE_OWNED`

Page conversions

DONATION	*Initiator*	*Completer*
Before	`PKVM_PAGE_OWNED`	`PKVM_NOPAGE`
After	`PKVM_NOPAGE`	`PKVM_PAGE_OWNED`

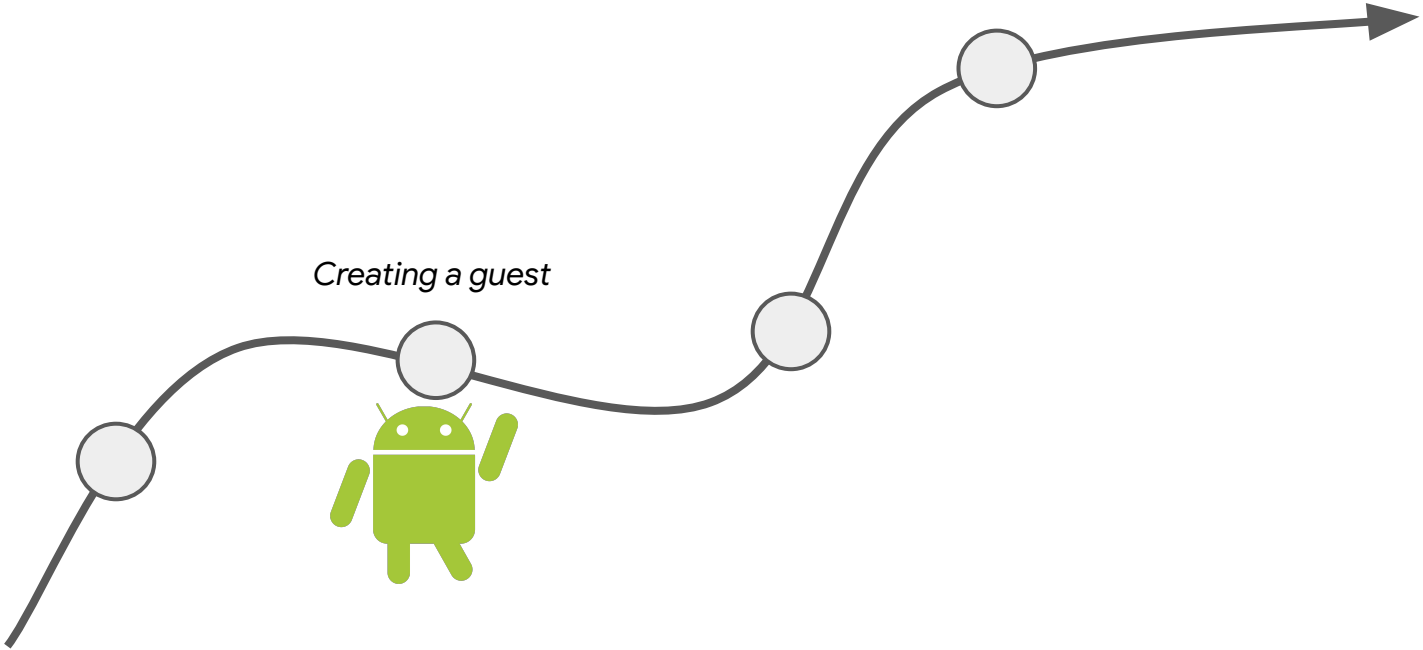
SHARE	*Initiator*	*Completer*
Before	`PKVM_PAGE_OWNED`	`PKVM_NOPAGE`
After	`PKVM_PAGE_SHARED_OWNED`	`PKVM_PAGE_SHARED_BORROWED`

Page conversions

DONATION	*Initiator*	*Completer*
Before	`PKVM_PAGE_OWNED`	`PKVM_NOPAGE`
After	`PKVM_NOPAGE`	`PKVM_PAGE_OWNED`

SHARE	*Initiator*	*Completer*
Before	`PKVM_PAGE_OWNED`	`PKVM_NOPAGE`
After	`PKVM_PAGE_SHARED_OWNED`	`PKVM_PAGE_SHARED_BORROWED`

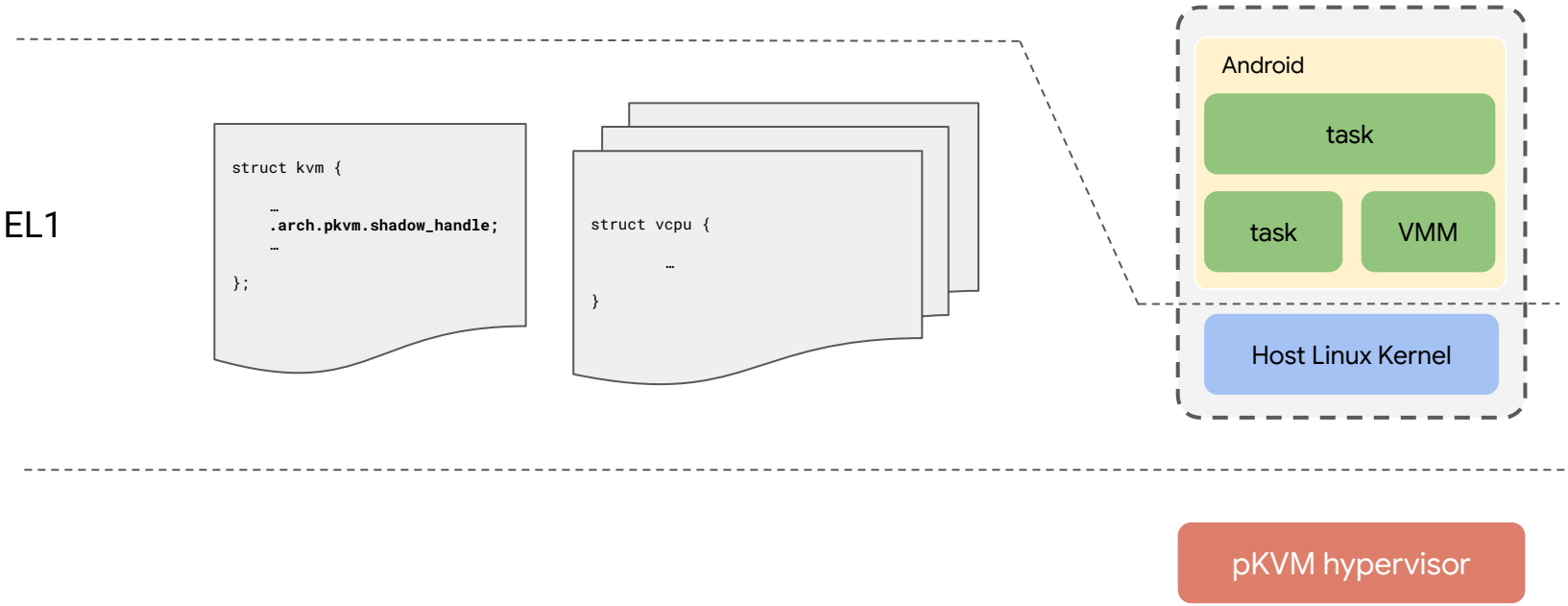
UNSHARE	*Initiator*	*Completer*
Before	`PKVM_PAGE_SHARED_OWNED`	`PKVM_PAGE_SHARED_BORROWED`
After	`PKVM_PAGE_OWNED`	`PKVM_NOPAGE`


Creating a guest

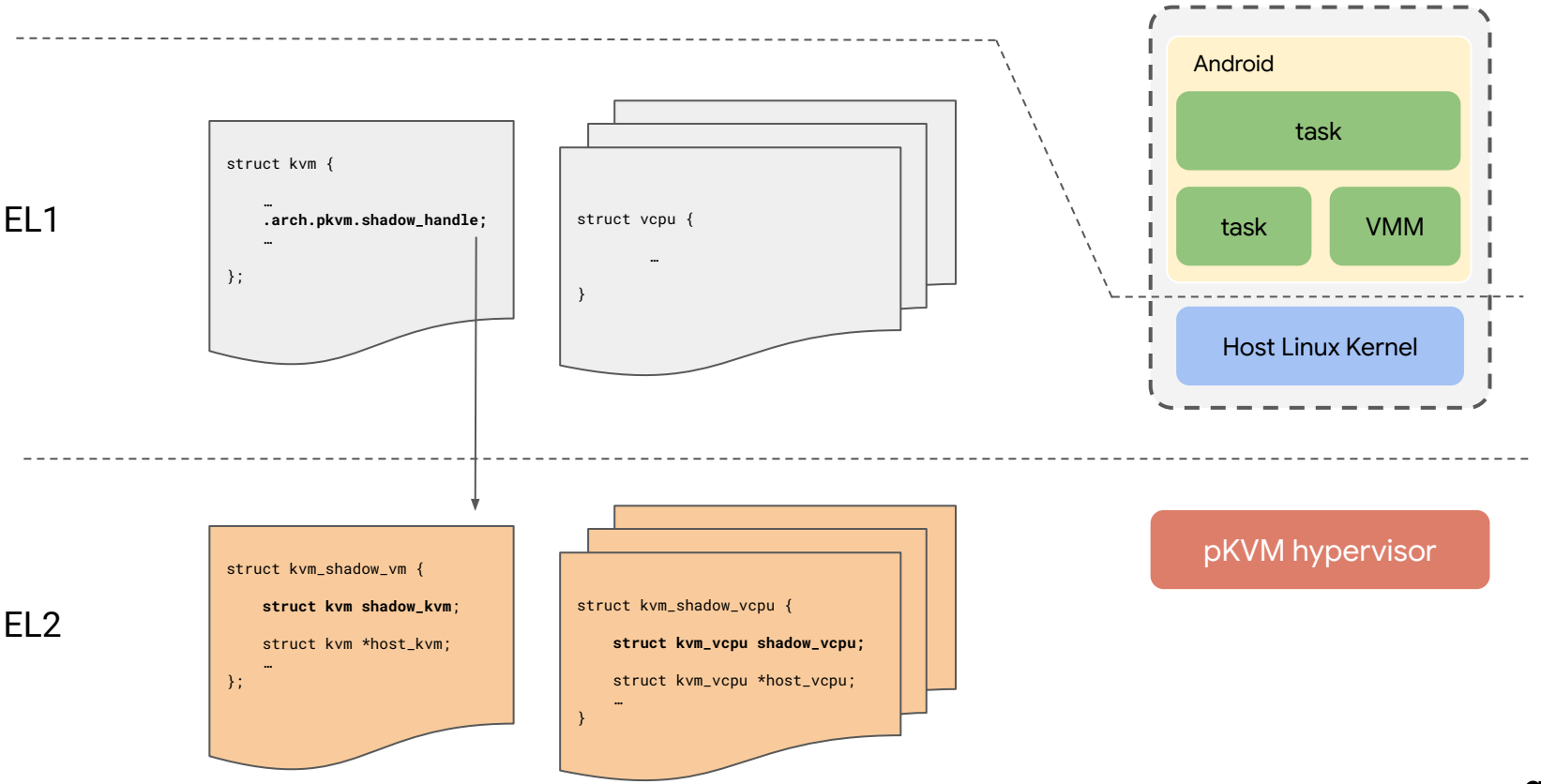
Creating a guest

- Host **allocates pages** for EL2 (`GFP_KERNEL_ACCOUNT`)
- Hypervisor receives a __pkvm_init_shadow() hypercall and
  - **Converts** allocated pages with a *host-to-hypervisor* donation
  - Allocates a **shadow_handle**
  - Initializes EL2-private guest and vCPU state (`struct kvm`, `struct kvm_vcpu`, stage-2 PGD)
- The shadow_handle is returned to EL1 and stored struct kvm

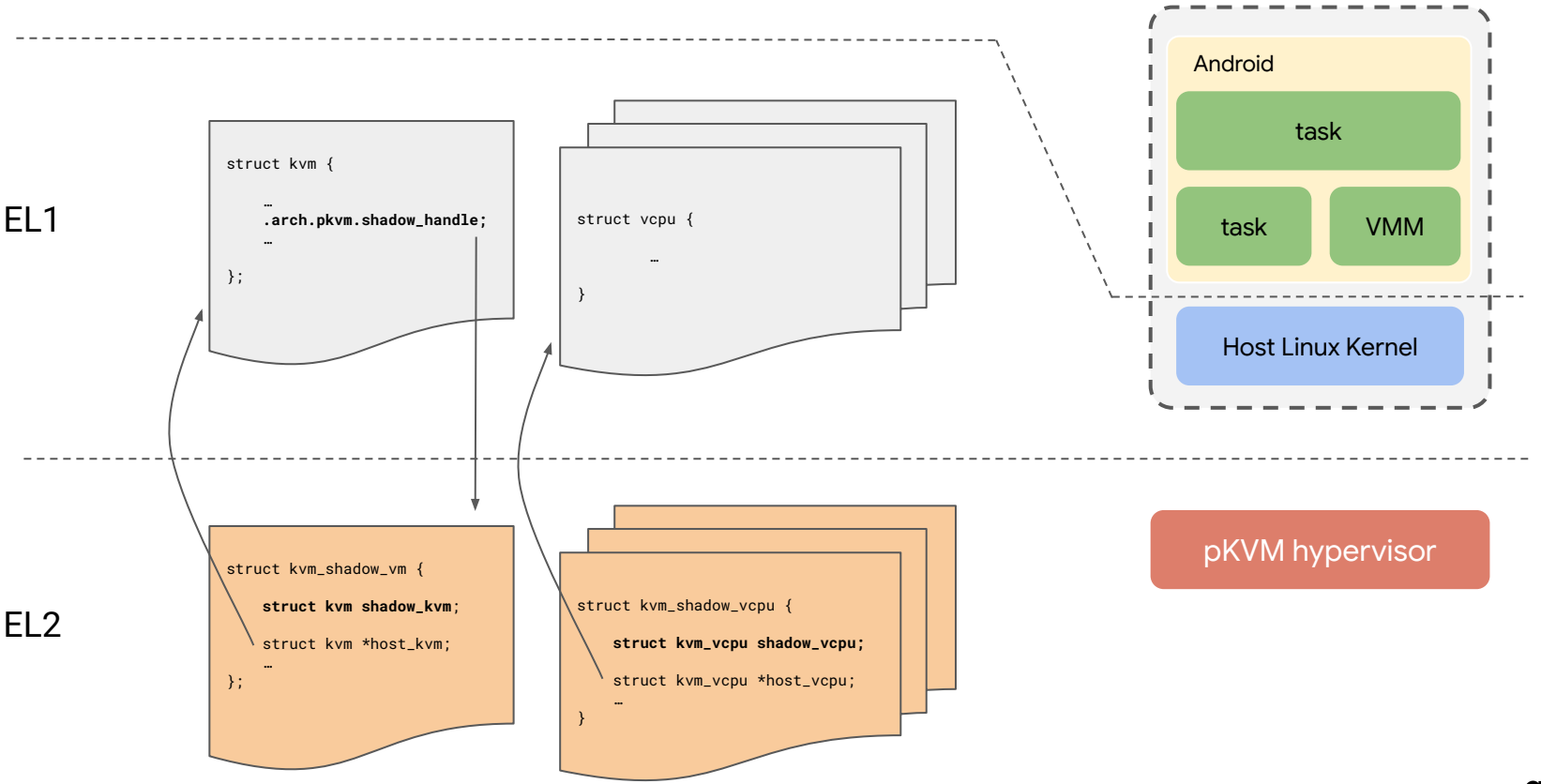
Guest data-structures



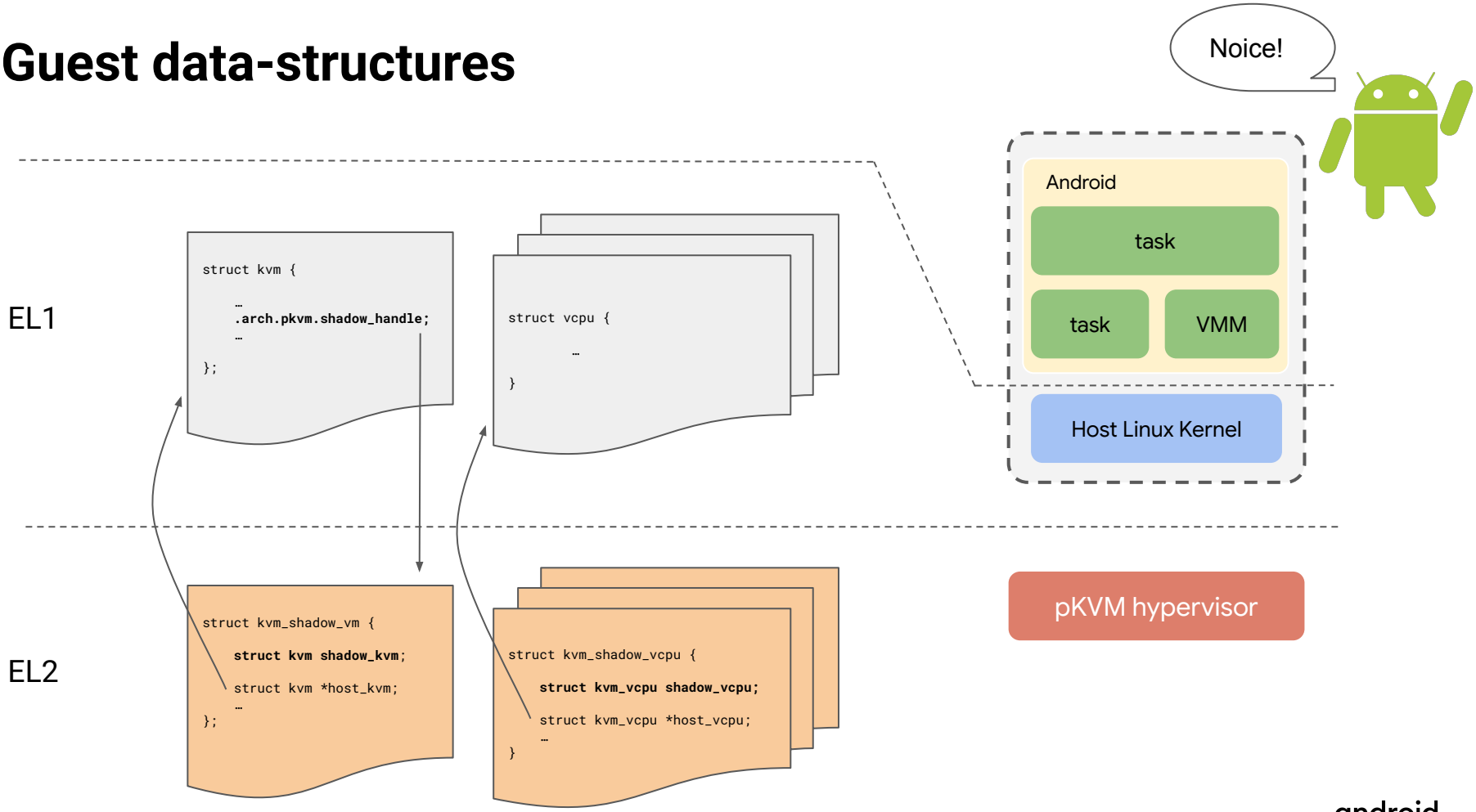
Guest data-structures



Guest data-structures

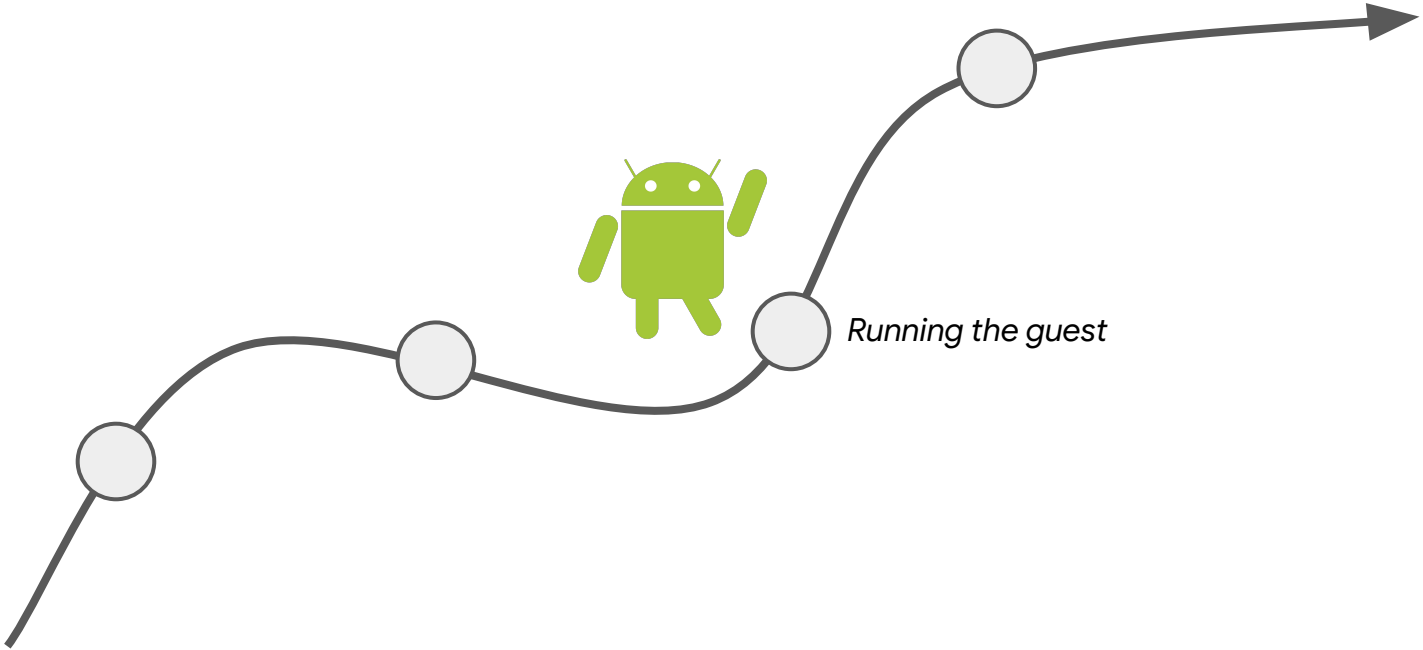


Guest data-structures



EL2 infrastructure

- hyp_spin_lock() / hyp_spin_unlock()
  - No mutex, EL2 is non-preemptible
- CONFIG_NVHE_EL2_DEBUG
  - hyp_lock_assert_held()
- Buddy page allocator (limited usage)
- per-cpu variables
- hyp_vmemmap
- percpu fixmap
- tracing (WiP)
- …


Running the guest

vCPU load/put

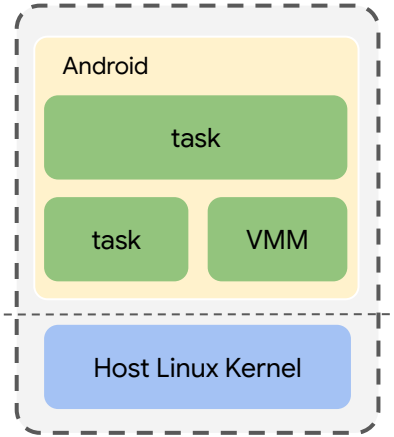
- KVM/arm64 uses `vcpu_load()/vcpu_put()` optimization to make a vCPU “resident”
- Host issues a `__pkvm_vcpu_load(shadow_handle, vcpu_id, …)` hypercall
  - Hypervisor **sanity checks** parameters
  - It then **takes a reference** on the shadow VM (to prevent teardown while in use)
  - A per-cpu EL2 variable is updated to point at loaded shadow
- Subsequent hypercalls (e.g. `__vcpu_run()`) may **require a loaded vCPU**
- `__pkvm_vcpu_put()` will ***“sync”*** the shadow vCPU state with the host for non-protected VMs, drop the reference on the shadow, and clear the EL2 per-cpu variable

Sync and flush

EL1

```
struct kvm {
    …
    .arch.pkvm.shadow_handle;
    …
};
```

```
struct vcpu {
        …
}
```



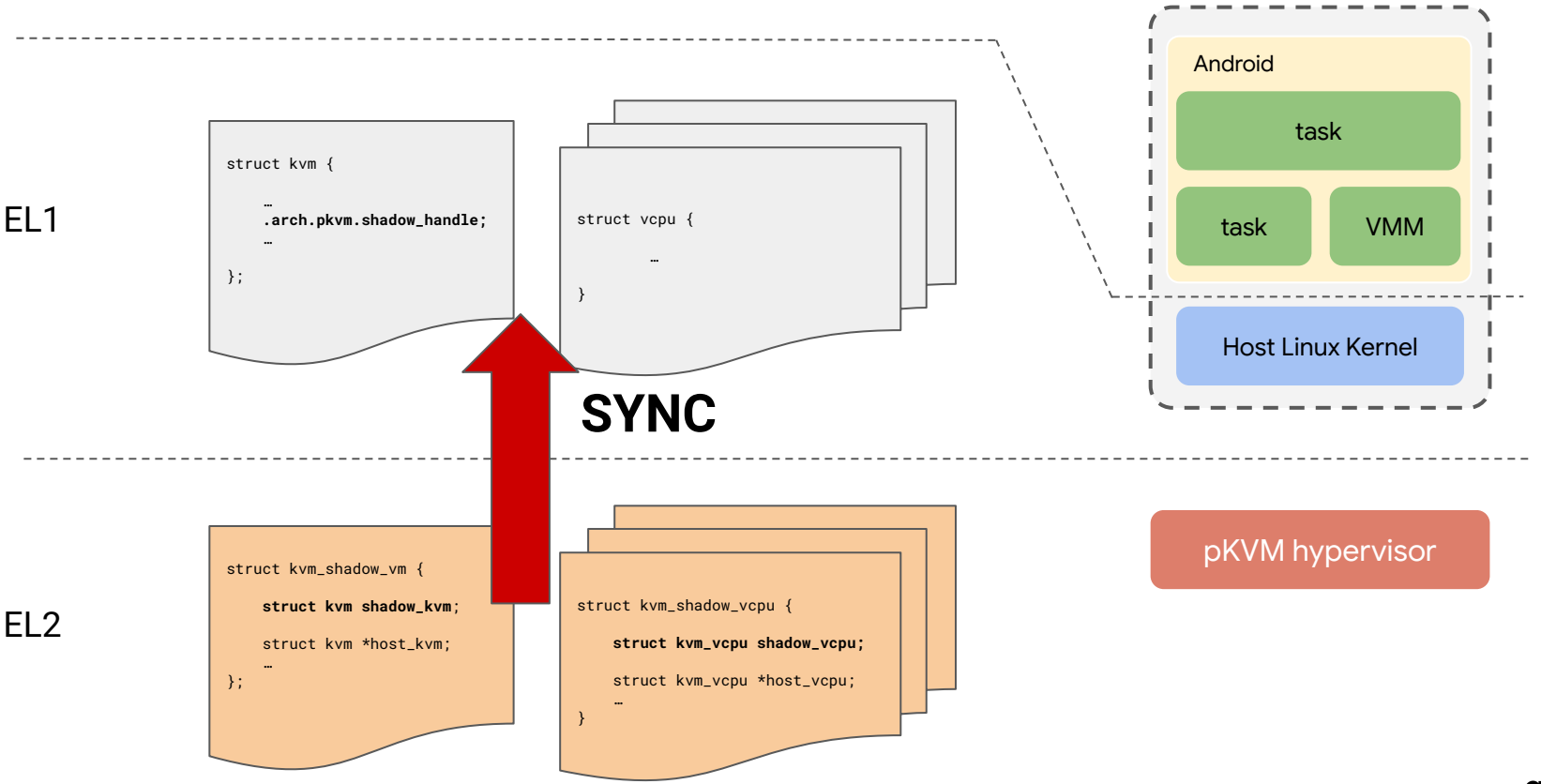
EL2

```
struct kvm_shadow_vm {
    struct kvm shadow_kvm;
    struct kvm *host_kvm;
    …
};
```

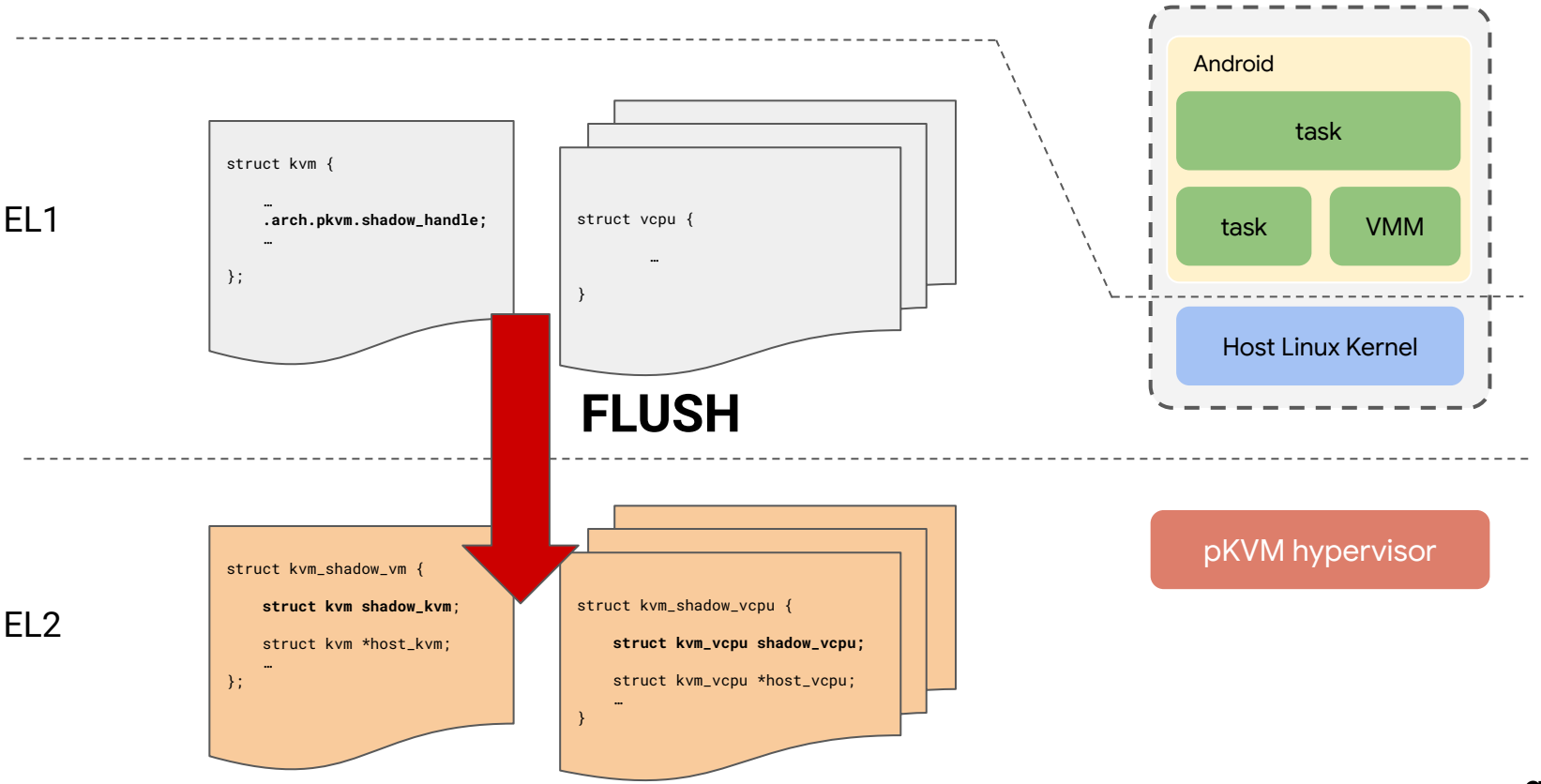
```
struct kvm_shadow_vcpu {
    struct kvm_vcpu shadow_vcpu;
    struct kvm_vcpu *host_vcpu;
    …
}
```

pKVM hypervisor

Sync and flush



Sync and flush



vCPU run

- Host issues a `__vcpu_run()` hypercall
  - Hypervisor expects a loaded vCPU
  - Hypervisor ***"flushes"*** vCPU state
  - Context switches using the **shadow vCPU**, and erets in the guest
  - On exit, it switches back to the host, and "***sync***" the state

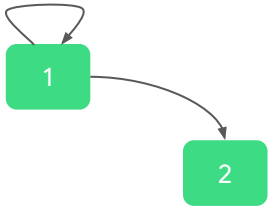
Exit handling

- Some exits can be **handled at EL2 directly**, for example
  - FP traps requiring to context switch from host FP state to the guest's
  - Some of the vgic_v3 sysreg emulation
  - PAuth traps
  - Some guest hypercalls
  - …
- Other exits **need handling on the host** side
- Host-side handling can be similar to standard KVM, with a notable exception for **instruction and data aborts**

Instruction and data aborts

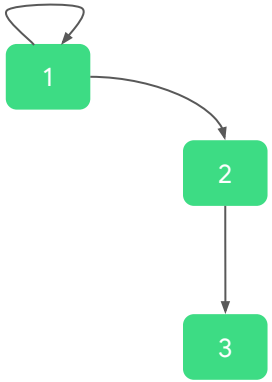
Instruction and data aborts

- Instruction/data abort
- **ESR and GPA** copied to host vcpu struct
- “`eret`” to EL1



Instruction and data aborts

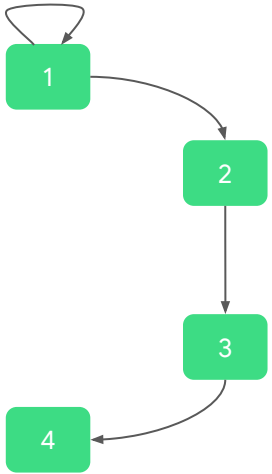
- Instruction/data abort
- **ESR and GPA** copied to host vcpu struct
- "`eret`" to EL1



- GPA is converted to HVA
- KVM takes ***long-term GUP pin***
  - *Prevents swap and page migration*
  - *More on this later*

Instruction and data aborts

- Instruction/data abort
- **ESR and GPA** copied to host vcpu struct
- "`eret`" to EL1

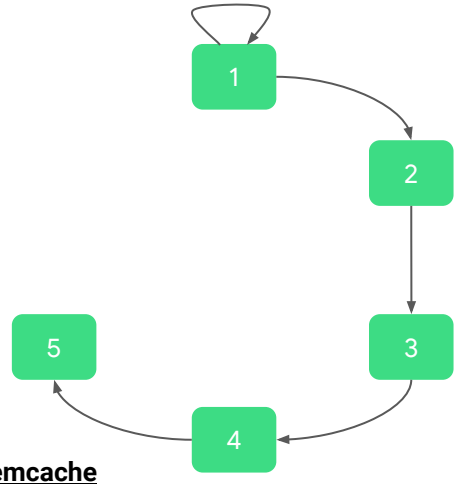


- GPA is converted to HVA
- KVM takes ***long-term GUP pin***
  - *Prevents swap and page migration*
  - *More on this later*

- KVM tops-up per-vcpu memcache
- `__pkvm_guest_map(pfn, gfn, …)` hypercall

Instruction and data aborts

- Instruction/data abort
- **ESR and GPA** copied to host vcpu struct
- "`eret`" to EL1



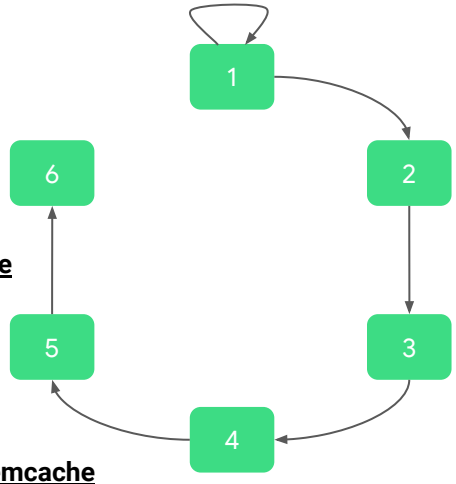
- GPA is converted to HVA
- KVM takes ***long-term GUP pin***
  - *Prevents swap and page migration*
  - *More on this later*

- Hypervisor tops up its **own vcpu memcache**
- Each page goes through a *host-to-hypervisor* donation

- KVM tops-up per-vcpu memcache
- `__pkvm_guest_map(pfn, gfn, …)` hypercall

Instruction and data aborts

- Instruction/data abort
- **ESR and GPA** copied to host vcpu struct
- "`eret`" to EL1



- For **protected** guests: host-to-guest **donation**
- For **non-protected** guests: host-to-guest **share**

- GPA is converted to HVA
- KVM takes ***long-term GUP pin***
  - *Prevents swap and page migration*
  - *More on this later*

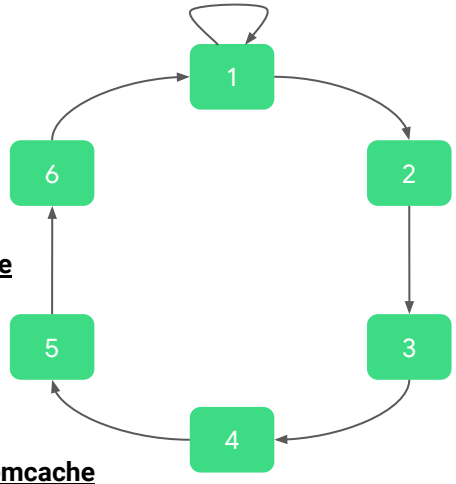
- Hypervisor tops up its **own vcpu memcache**
- Each page goes through a *host-to-hypervisor* donation

- KVM tops-up per-vcpu memcache
- `__pkvm_guest_map(pfn, gfn, …)` hypercall

Instruction and data aborts

- Instruction/data abort
- **ESR and GPA** copied to host vcpu struct
- "`eret`" to EL1

- "`eret`" to the host
- `__vcpu_run()` hypercall



- For **protected** guests: host-to-guest **donation**
- For **non-protected** guests: host-to-guest **share**

- GPA is converted to HVA
- KVM takes ***long-term GUP pin***
  - *Prevents swap and page migration*
  - *More on this later*

- Hypervisor tops up its **own vcpu memcache**
- Each page goes through a *host-to-hypervisor* donation

- KVM tops-up per-vcpu memcache
- `__pkvm_guest_map(pfn, gfn, …)` hypercall

MMIO exits

- Hypervisor has no understanding of **memslots**
- Guest GPRs must remain **private** *(for protected guests)*

MMIO exits

- Hypervisor has no understanding of **memslots**
- Guest GPRs must remain **private** *(for protected guests)*

Not good for MMIO exits!

MMIO exits

- Hypervisor has no understanding of **memslots**
- Guest GPRs must remain **private** *(for protected guests)*

Not good for MMIO exits!

- Hypervisor exposes `ARM_SMCCC_KVM_FUNC_MMIO_GUARD_*` hypercalls
- Protected guests must ‘declare’ the **MMIO ranges** in their IPA space
- Hypervisor uses `r0` as a **transfer register** in these regions

Share hypercalls from guests

- pKVM exposes `ARM_SMCCC_KVM_FUNC_MEM_{UN}SHARE` hypercalls to protected guests
- If requested page is mapped in guest, hypervisor applies a **guest-to-host {un}share** conversion
- If not paged-in, "`eret`" with a fake ESR to trigger **the page fault path**. The guest PC is rewound by one instruction, to re-try on next vcpu_run

Notes on guest firmware loading

- Device bootloader (ABL) copies **guest bootloader in a reserved memory region**
- Said memory region is **unmapped** from host stage-2 at pKVM init time
- When pages are initially mapped into the guest (due to host donations), the hypervisor **copies guest bootloader on the fly**, but only for **protected VMs**
- The guest bootloader can then load and verify the payload
- The IPA range where guest firmware should be loaded is specified via an ioctl()

Handling of host stage-2 faults

- Host stage-2 mappings are created **lazily**
- When a fault is taken, the hypervisor walks the host's stage-2 page-table to **check the state of the page**
  - If the PTE is invalid, but the owner id encoded in bits [63-1] is the host's **we idmap the page** and return to the host
  - If the PTE is valid, we've probably **raced with another CPU** and return to the host
  - If the PTE is invalid, and the owner id is not the host's, we caught it red-handed **accessing private memory**

Handling of host stage-2 faults

- Host stage-2 mappings are created **lazily**
- When a fault is taken, the hypervisor walks the host's stage-2 page-table to **check the state of the page**
  - If the PTE is invalid, but the owner id encoded in bits [63-1] is the host's **we idmap the page** and return to the host
  - If the PTE is valid, we've probably **raced with another CPU** and return to the host
  - If the PTE is invalid, and the owner id is not the host's, we caught it red-handed **accessing private memory**

You shall not pass!

- The hypervisor **injects an exception** back in the host
- If the fault was taken from EL1, ESR_EL1 is munged to report a same-level fault
- We set ESR_EL1.S1PTW to allow the host's exception handler to **distinguish this fault** from a normal stage-1 fault
- If the fault was taken from EL0, the host's handler will **SEGV the userspace** process
- If the fault was taken from EL1, we might be in trouble…

Stage-2 fault taken from EL1

- No problem if taken from `uaccess()` functions
- Big problem if taken from e.g. `process_vm_readv()`
  - `strace`-ing a malicious VMM that passes guest private memory to a syscall **brings the machine down**
  - Alternative solution required before this can land upstream

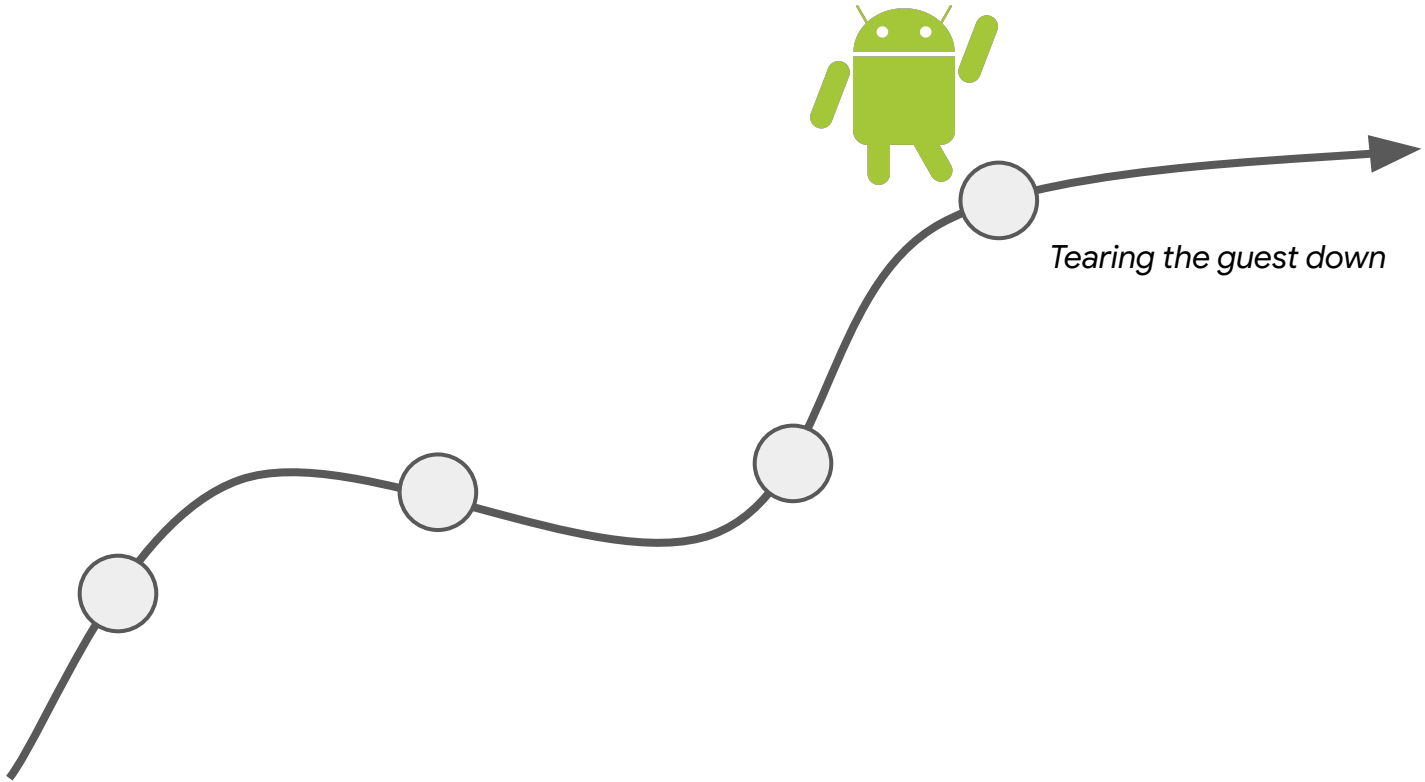
Promising solution

- Extend **private-fd** proposal [1] for pKVM
  - prevents swap and page-migrations
  - should prevent spurious kernel accesses to guest private memory by construction
  - offers a suitable API for hypervisor-assisted page migration in the future
- Support for **in-place conversions** is a must-have
- Would **secretmem** extended with the new memfile_notifier suffice?

[1] https://lore.kernel.org/lkml/20220706082016.2603916-1-chao.p.peng@linux.intel.com/

Not so promising solution

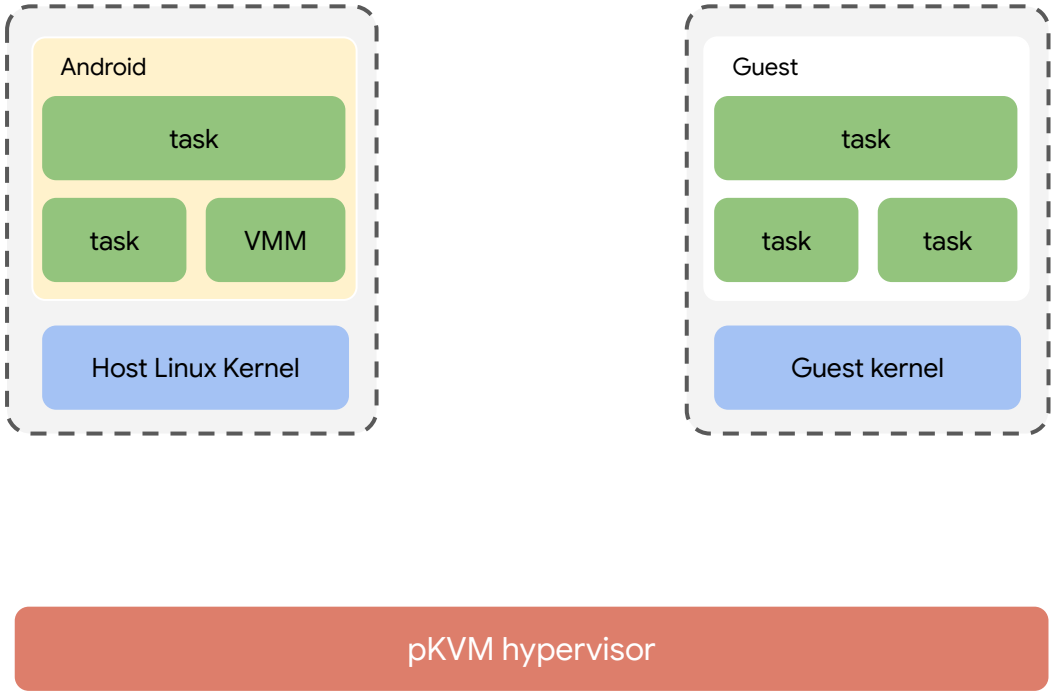
- Silently kill the **guest** at EL2, poison memory, and return to the host
- Pros:
  - Prevents host crashes
  - Longterm GUP pin sufficient (?)
- Cons:
  - Complexity at EL2
  - The guest is **incorrectly sanctioned**
  - KVM made aware **asynchronously**, hard to debug


Tearing the guest down

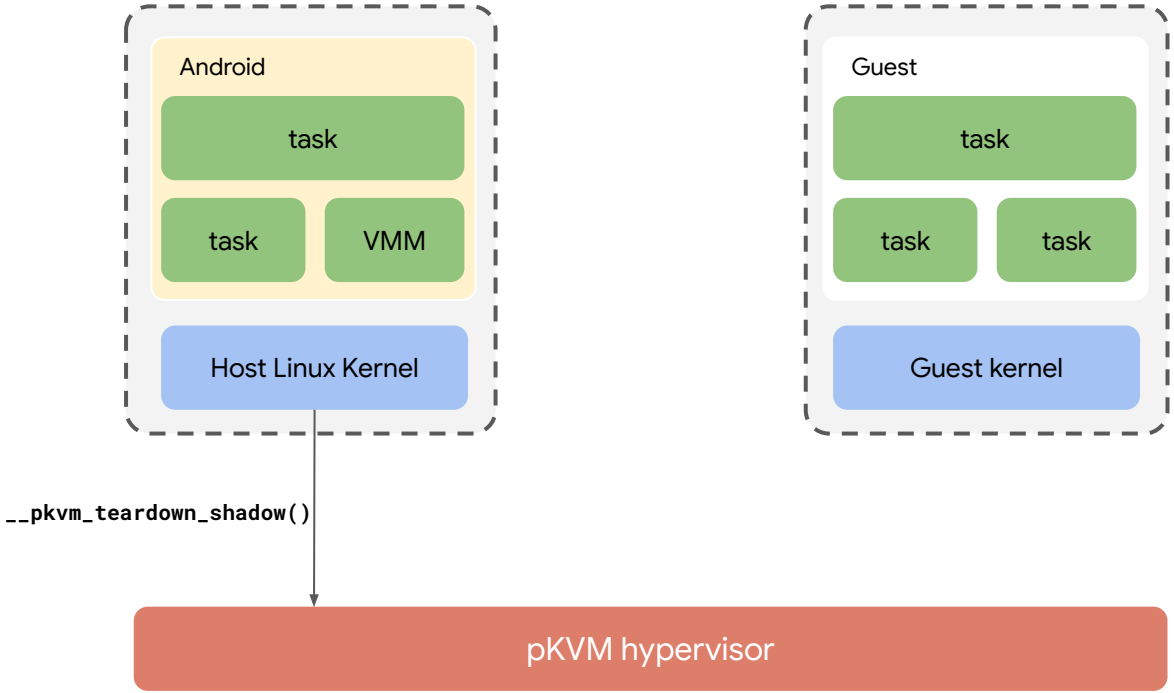
Guest teardown

- `__pkvm_teardown_shadow()` can be called when there are **no loaded vCPUs** for the guest
- Guest pages need to be returned to the host
  - Needs **poisoning**!
  - Reminder: EL2 is **non-preemptible**
  - Cannot be done in a single step

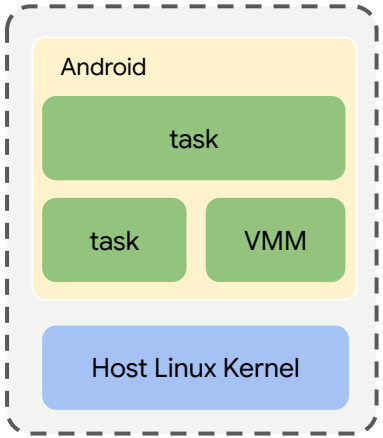
Reclaiming guest pages



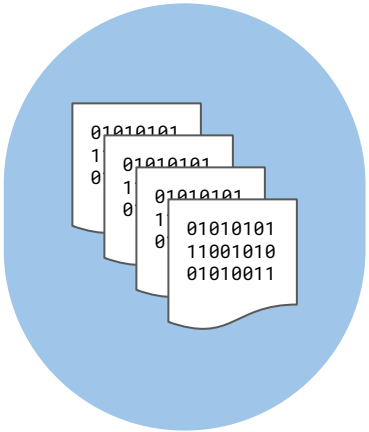
Reclaiming guest pages



Reclaiming guest pages

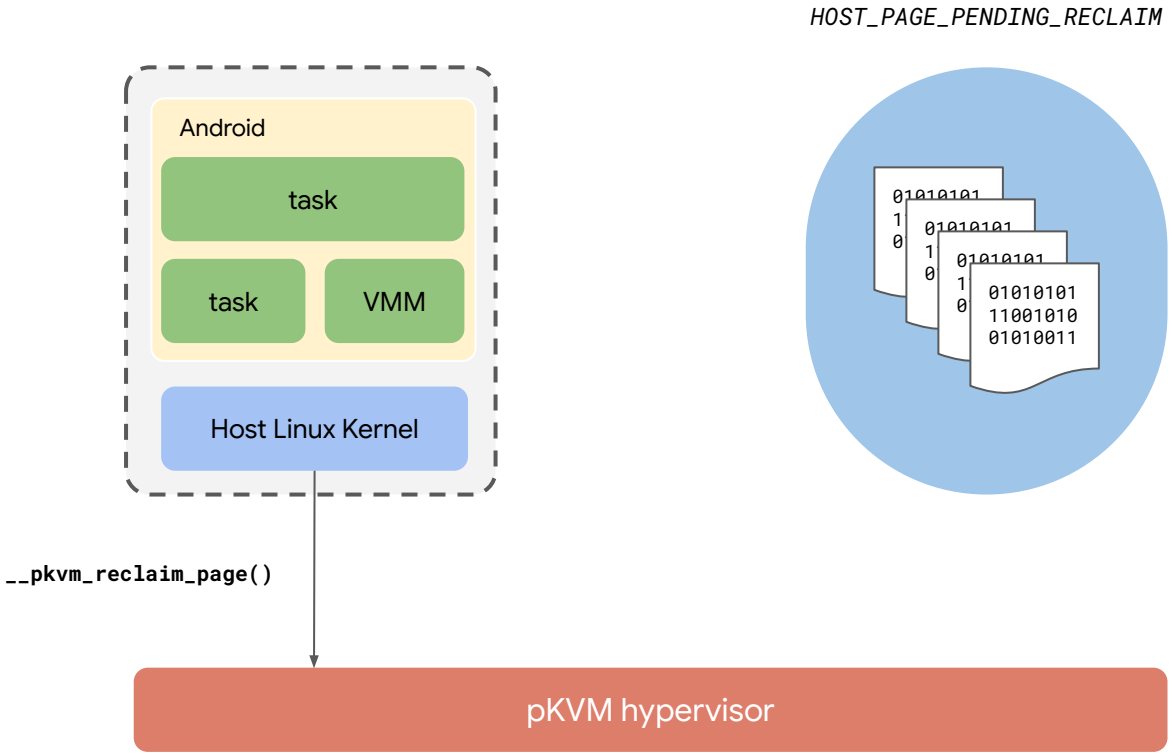


HOST_PAGE_PENDING_RECLAIM

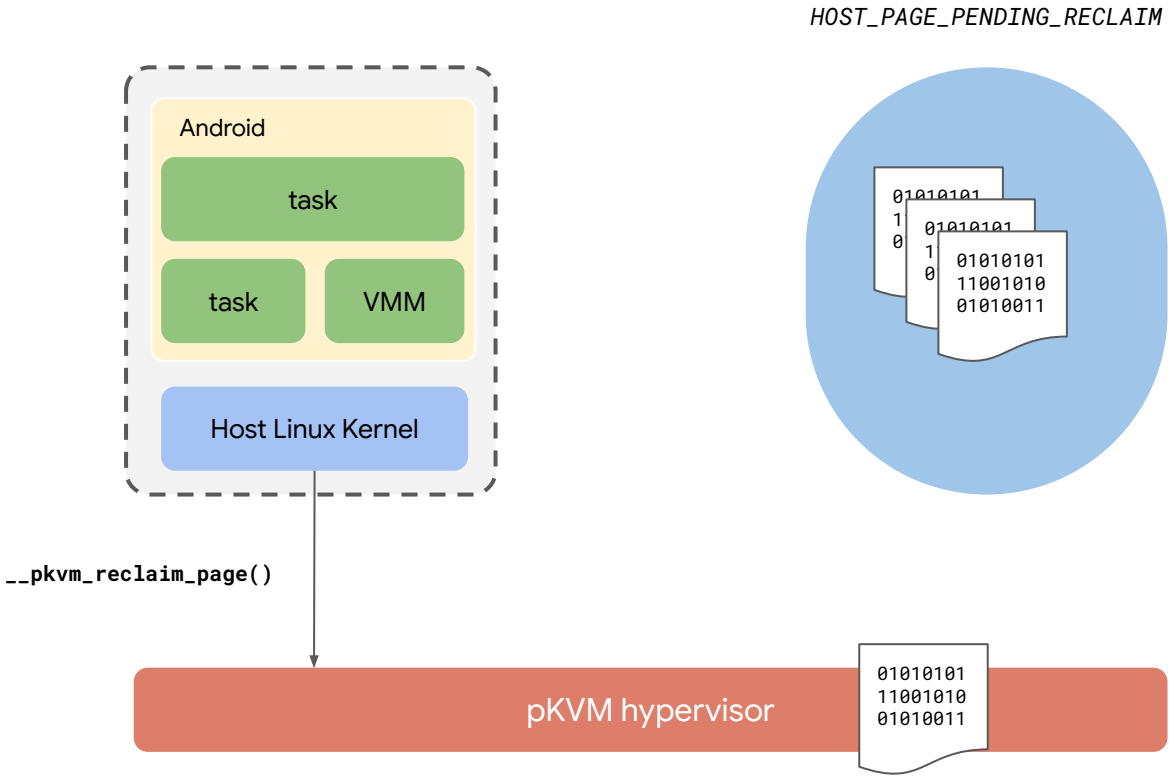


pKVM hypervisor

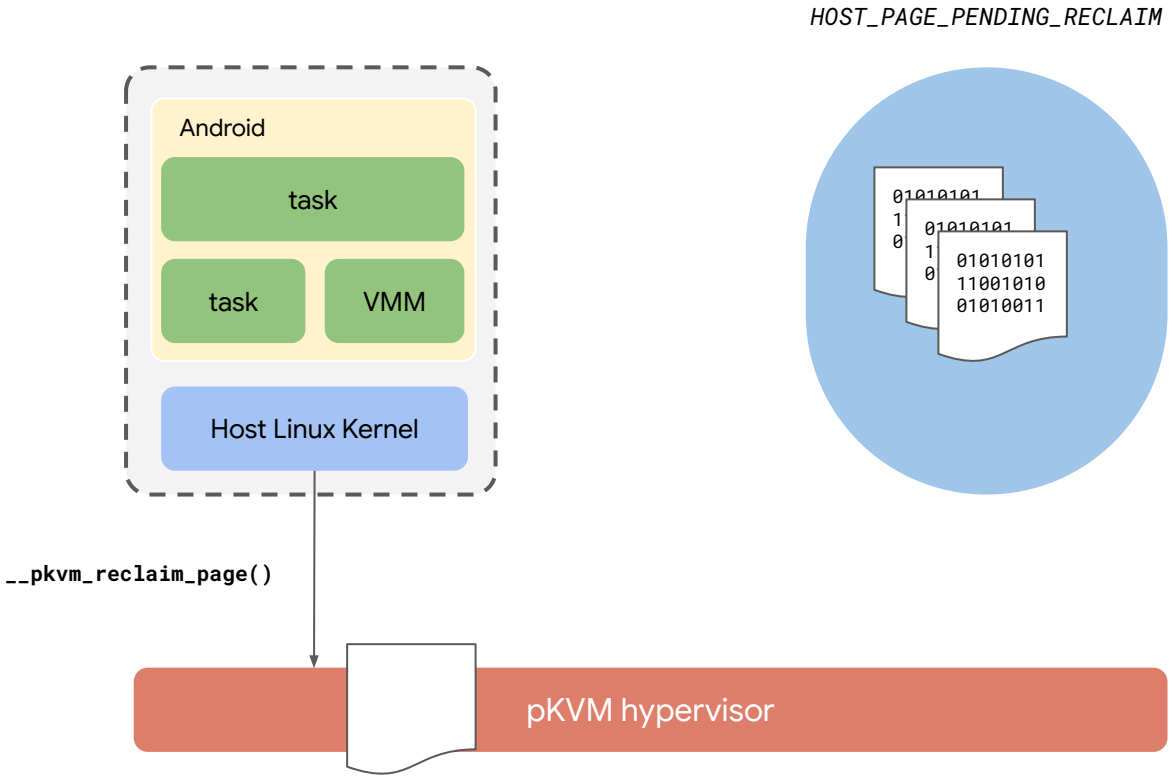
Reclaiming guest pages



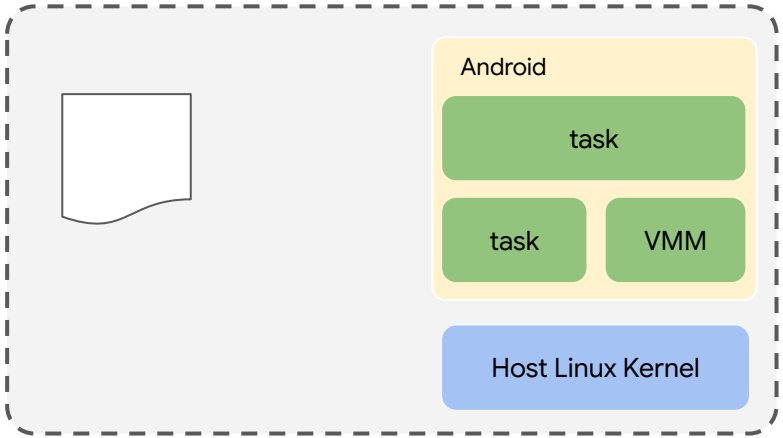
Reclaiming guest pages



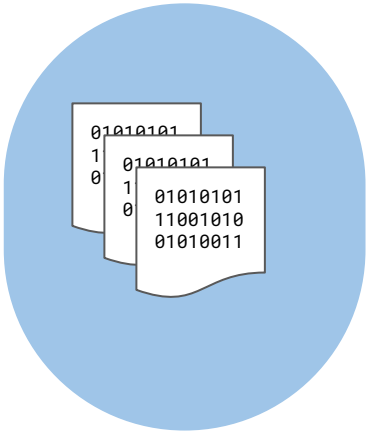
Reclaiming guest pages



Reclaiming guest pages

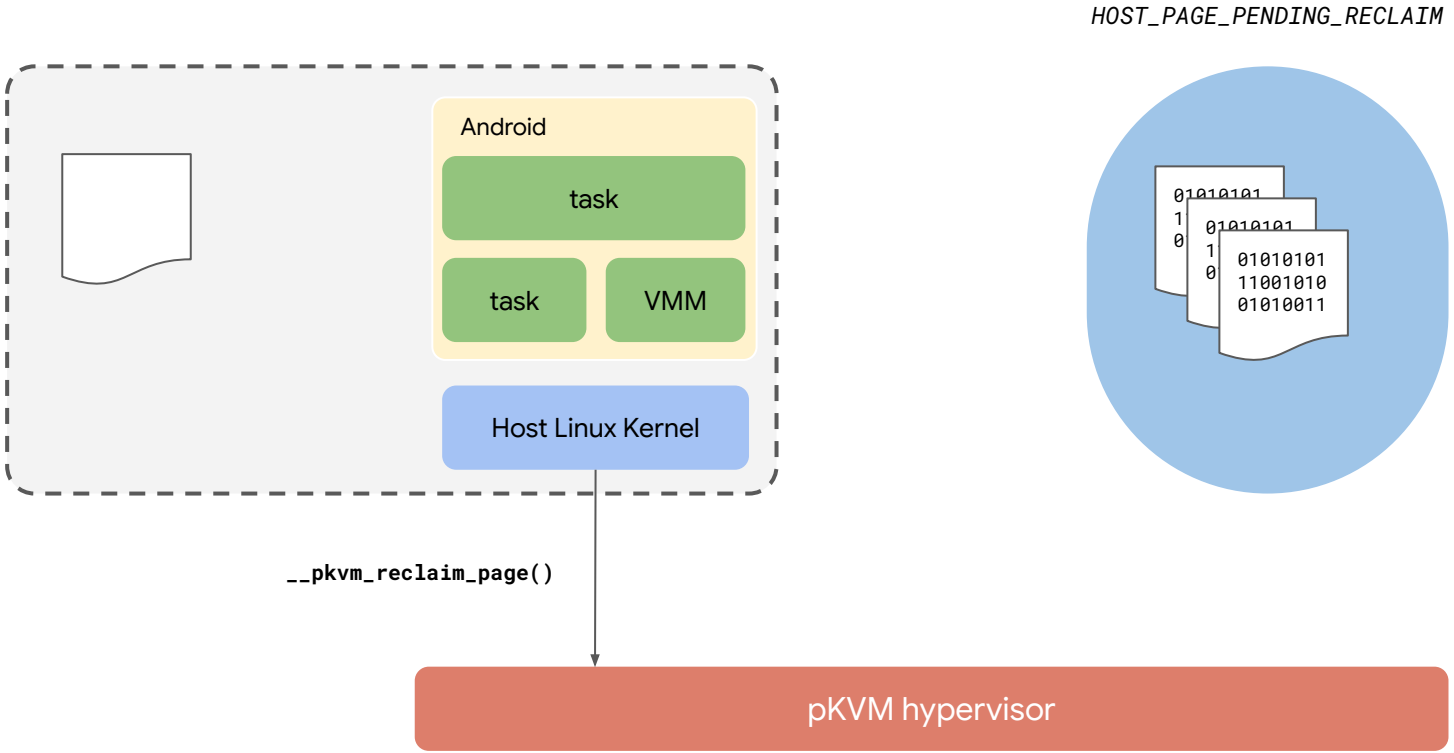


HOST_PAGE_PENDING_RECLAIM

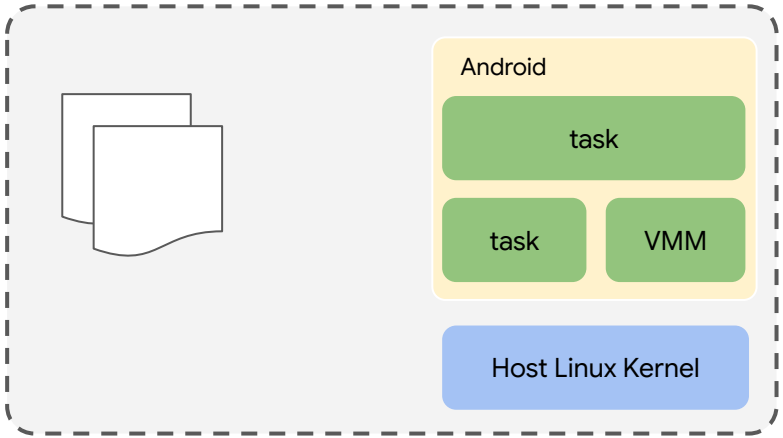


pKVM hypervisor

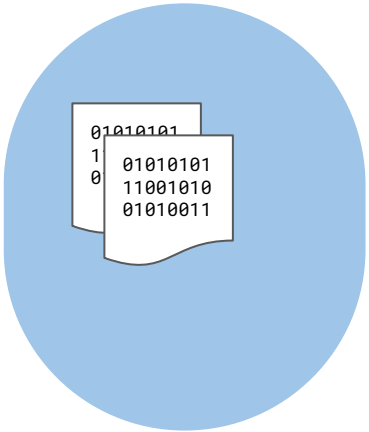
Reclaiming guest pages



Reclaiming guest pages

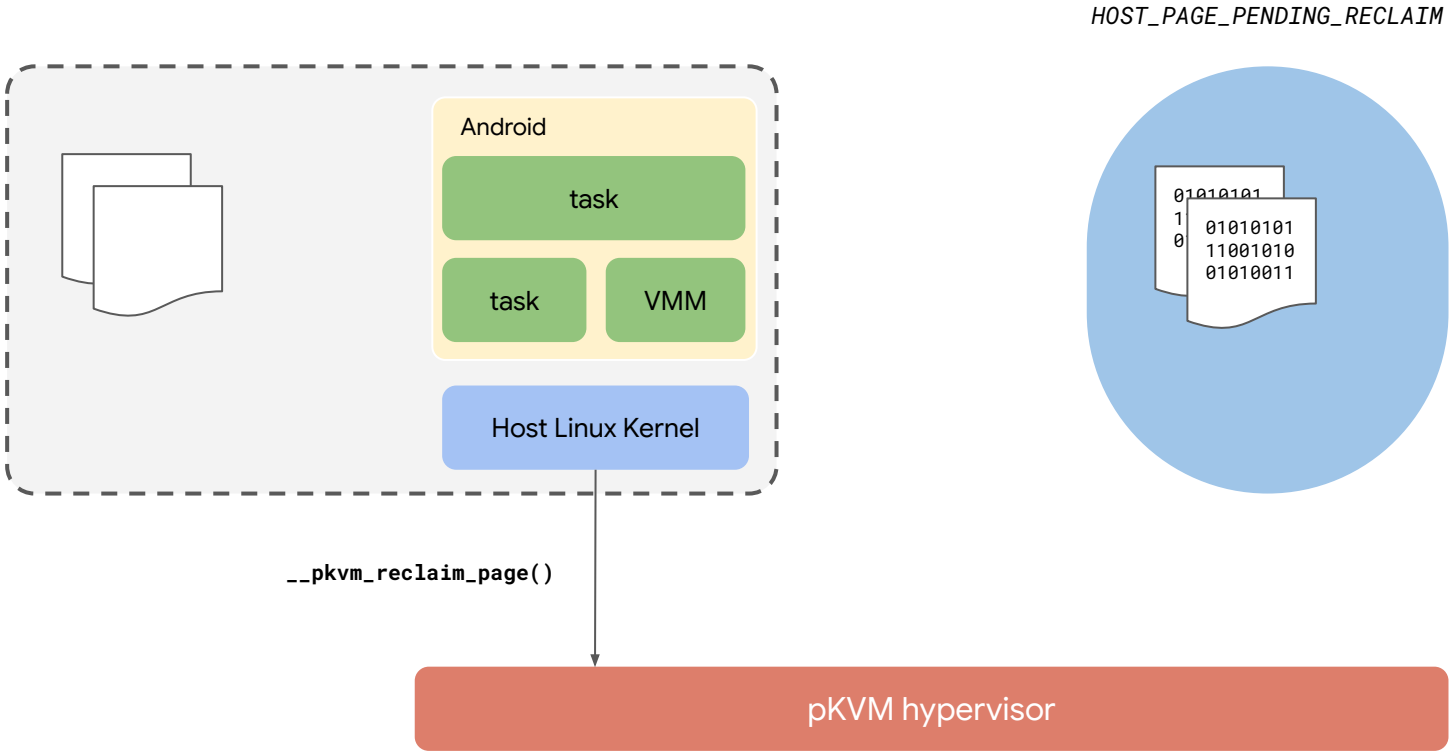


HOST_PAGE_PENDING_RECLAIM

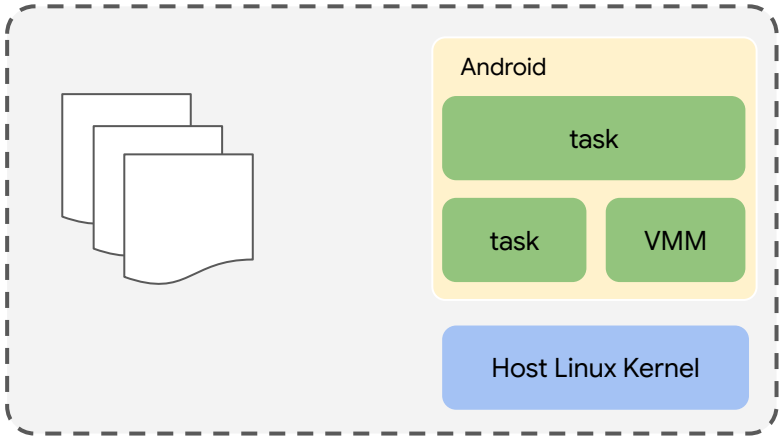


pKVM hypervisor

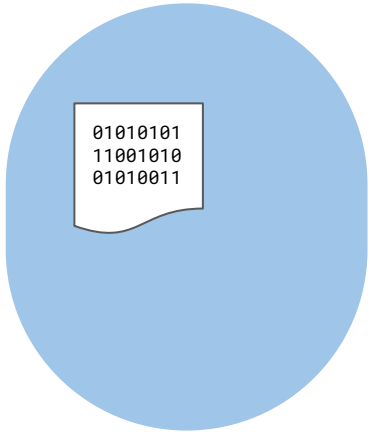
Reclaiming guest pages



Reclaiming guest pages

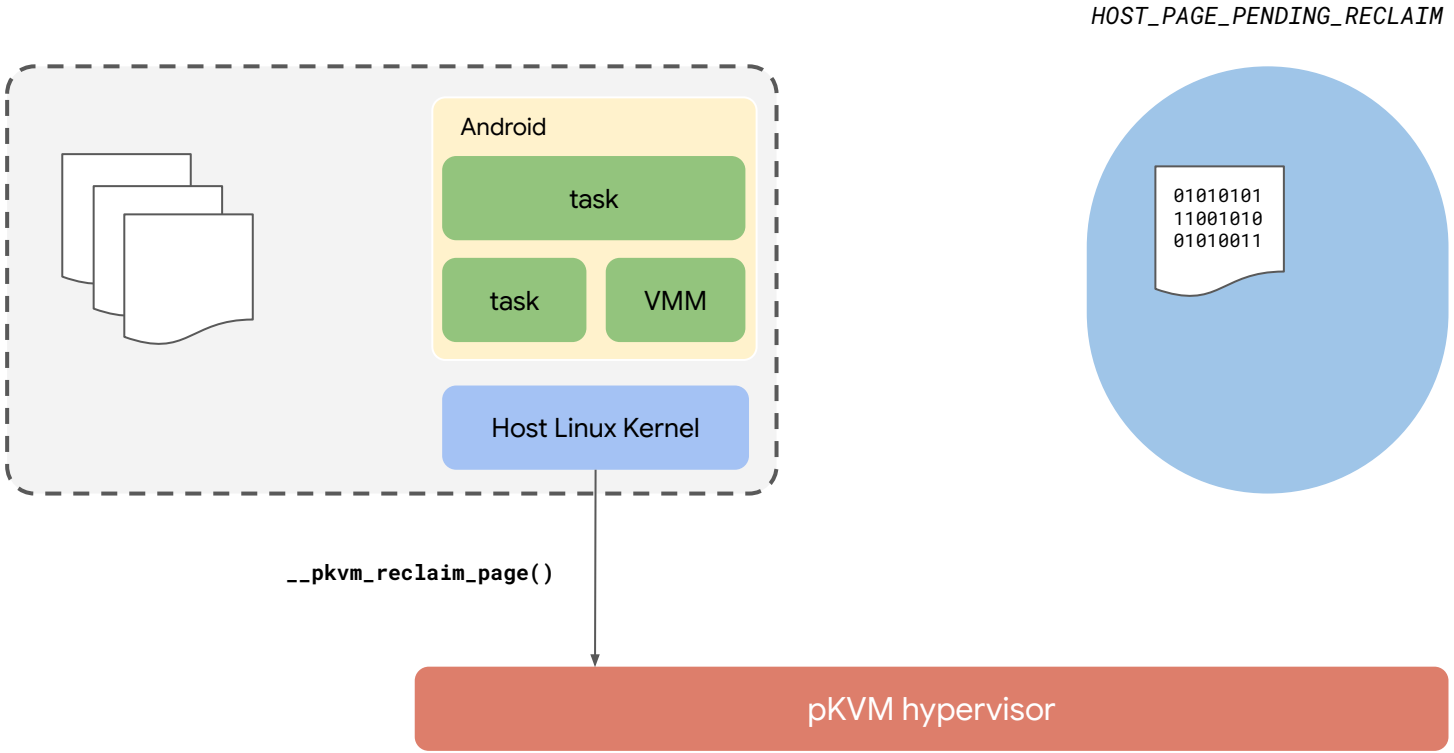


HOST_PAGE_PENDING_RECLAIM

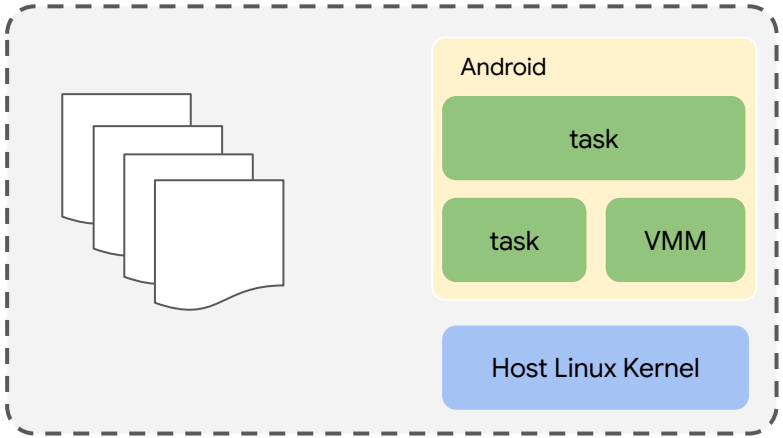


pKVM hypervisor

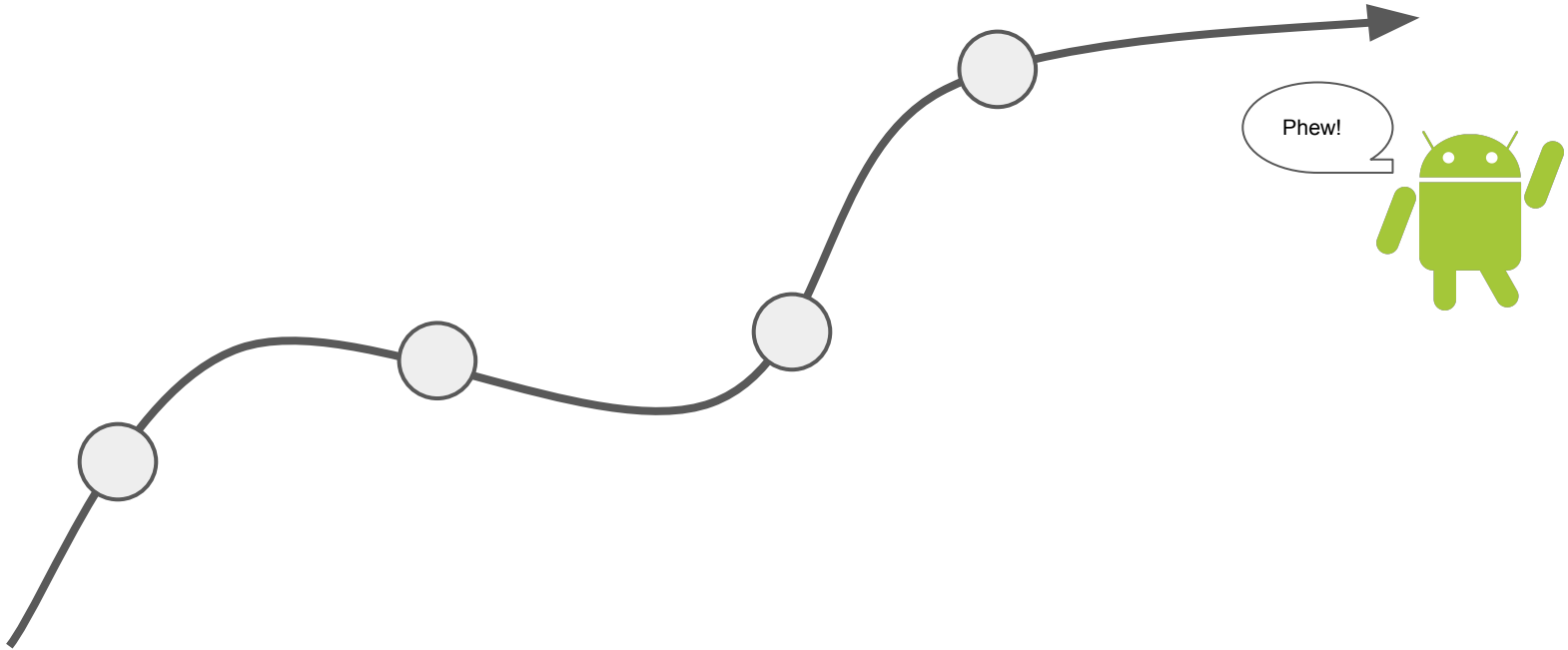
Reclaiming guest pages



Reclaiming guest pages

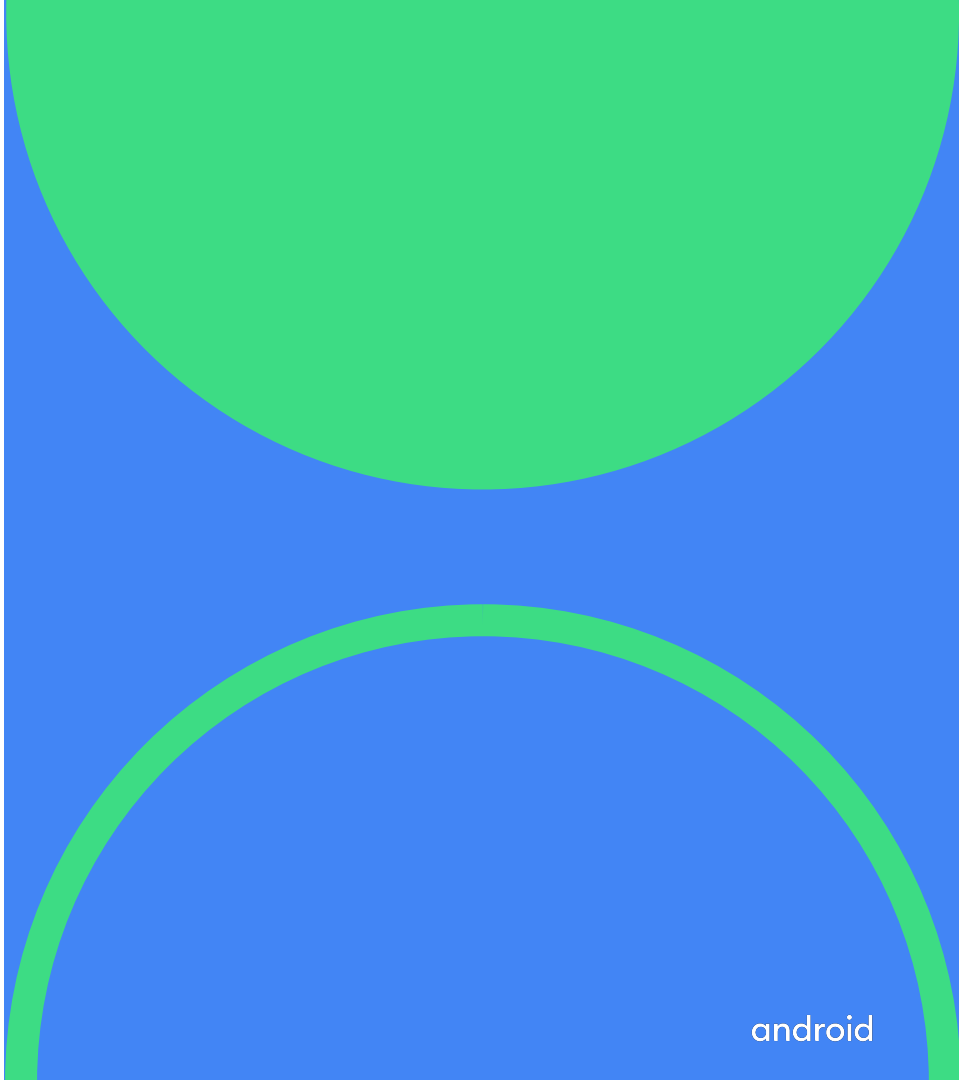


pKVM hypervisor


Phew!

Not covered in this talk

- FF-A / Secure-world communications
- IOMMUs / DMA protection
- Interrupts / timers / …
- PSCI
- TRNG
- ?



Limitations of current patch series

- Missing KVM features for non-protected guests (dirty-logging, RO memslots, …)
  - We have a working prototype
- Stubbed MMU notifiers (KSM, …) for non-protected guests
  - Requires minimal hypervisor support for multi-sharing
- No support for guest memory backed by huge-pages
- No support for kexec
- Device assignment

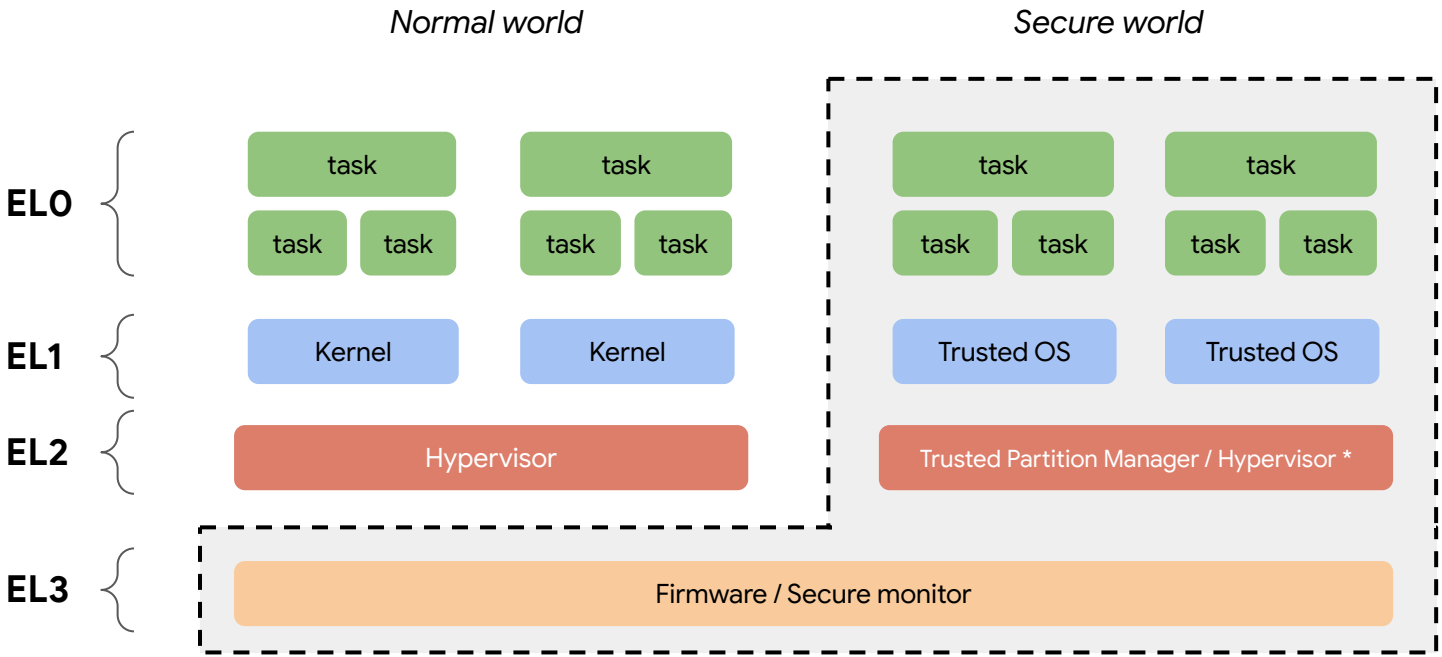
Thanks.

Please reach out! gperret@google.com / android-kvm@google.com

Questions?

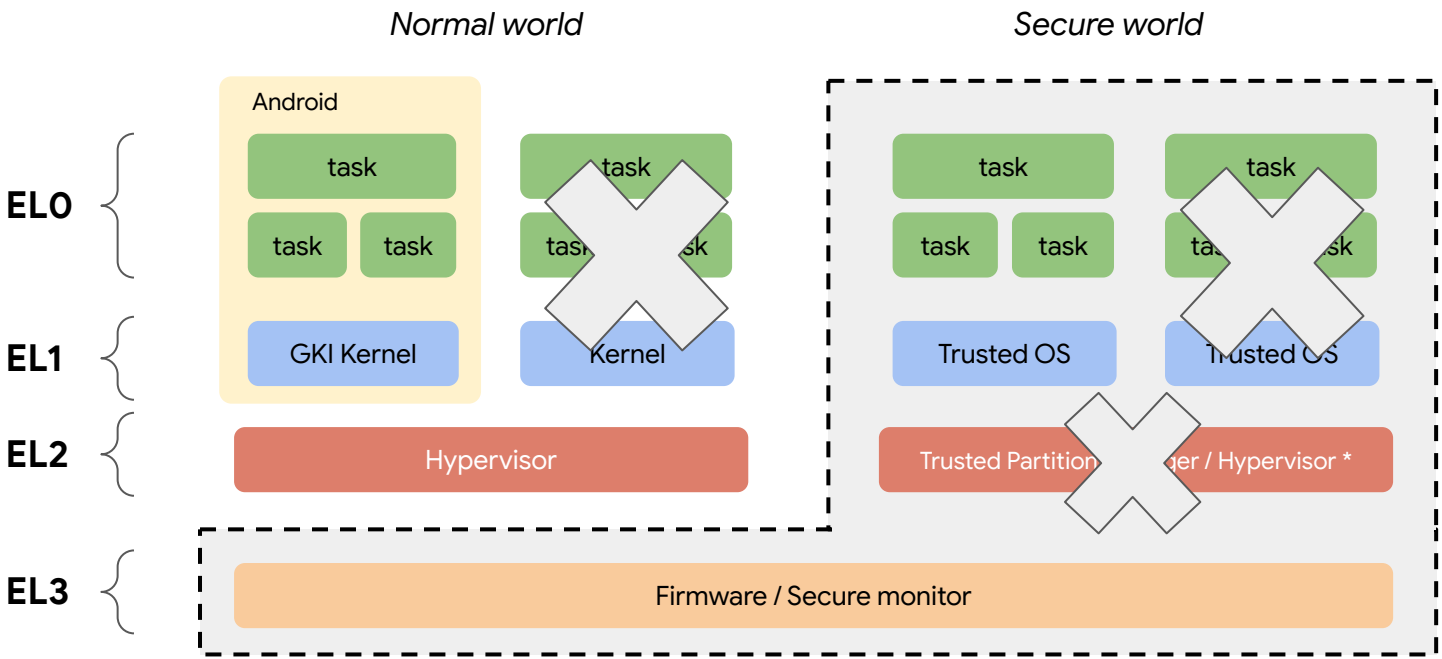
BACKUP

Exception levels on arm64, architecturally



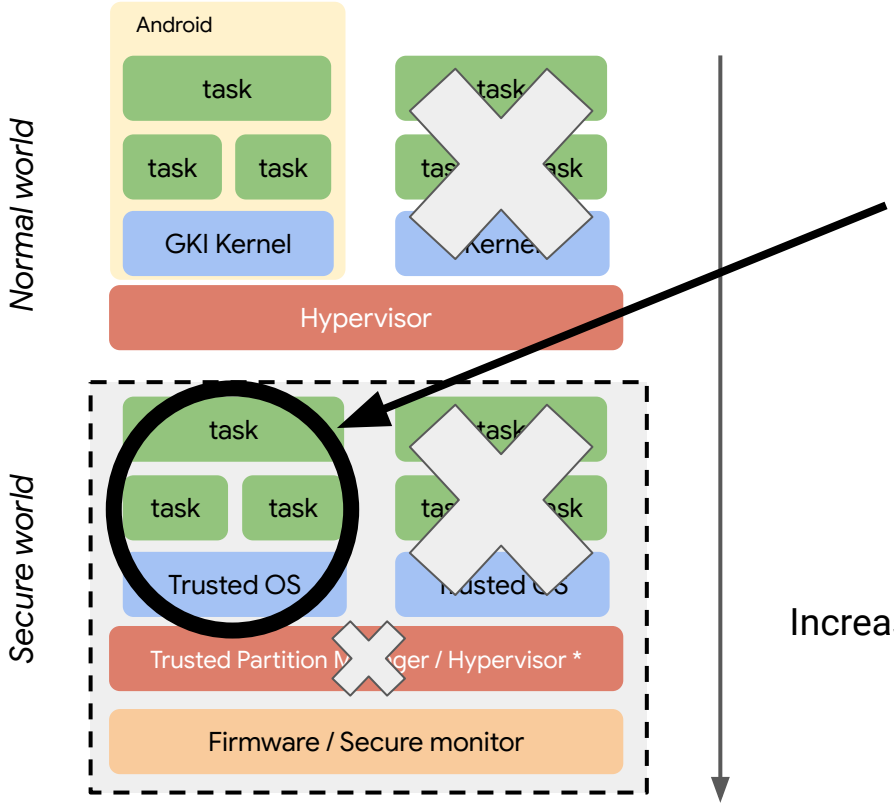
* From Arm v8.4A

Exception levels on arm64, in Android today



* From Arm v8.4A

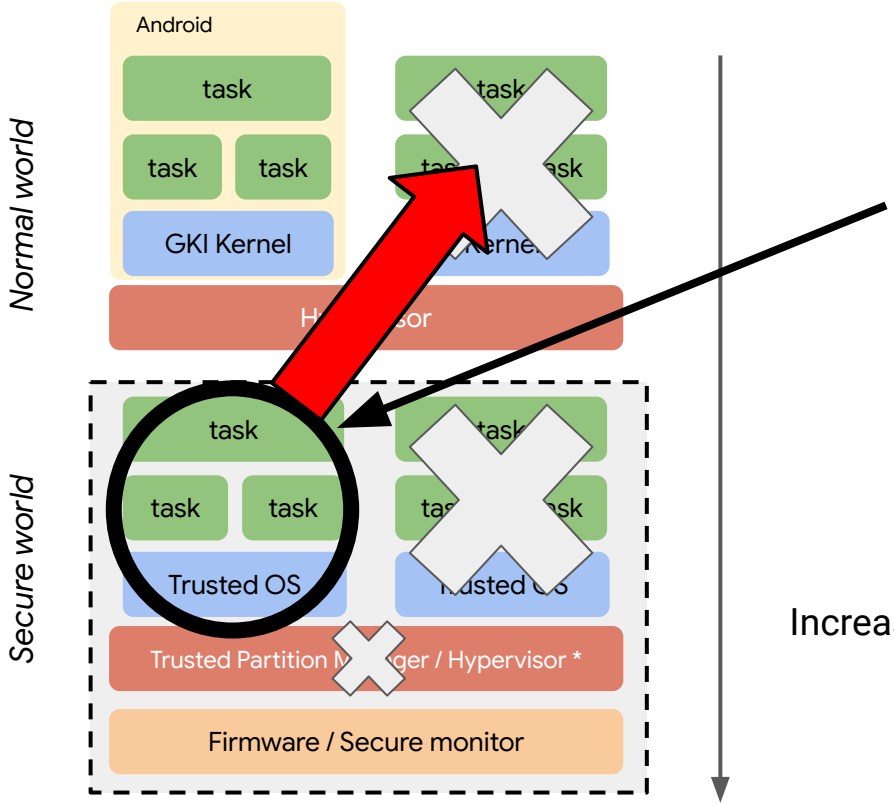
Exception levels on arm64, in Android today, by privilege



- DRM, crypto, ...
- Third party OSes
- Opaque blobs

Increasing privilege

Exception levels on arm64, in Android today, by privilege



- DRM, crypto, ...
- Third party OSes
- Opaque blobs

Increasing privilege

Guest creation

- Must happen **before the first vCPU run**
- The host allocates pages for EL2, and issues a `__pkvm_init_shadow()` hypercall
- Then the EL2 hypervisor:
  - **changes the owner** of allocated pages from “host” to “hypervisor” and unmaps/maps from corresponding page-tables (more details on ownership transitions later) ;
  - allocates a “**shadow handle**”, which represents the EL2 instance of the VM;
  - uses donated pages to store EL2-private struct kvm, struct kvm_vcpu, and associates them with the handle;
  - initializes the structs using e.g. **vCPU reset state**
  - checks and “pins” pages containing the host kvm and vcpu structs in a “shared” state;
  - allocates and initializes the guest’s **stage-2 PGD** using donated pages
  - returns the shadow handle to EL1
- EL1 stores the shadow handle in it’s own kvm struct
- Subsequent pKVM-specific hypercalls (e.g. `__vcpu_run()`) will **require a shadow handle**
- The above is required for protected **and** non-protected guests

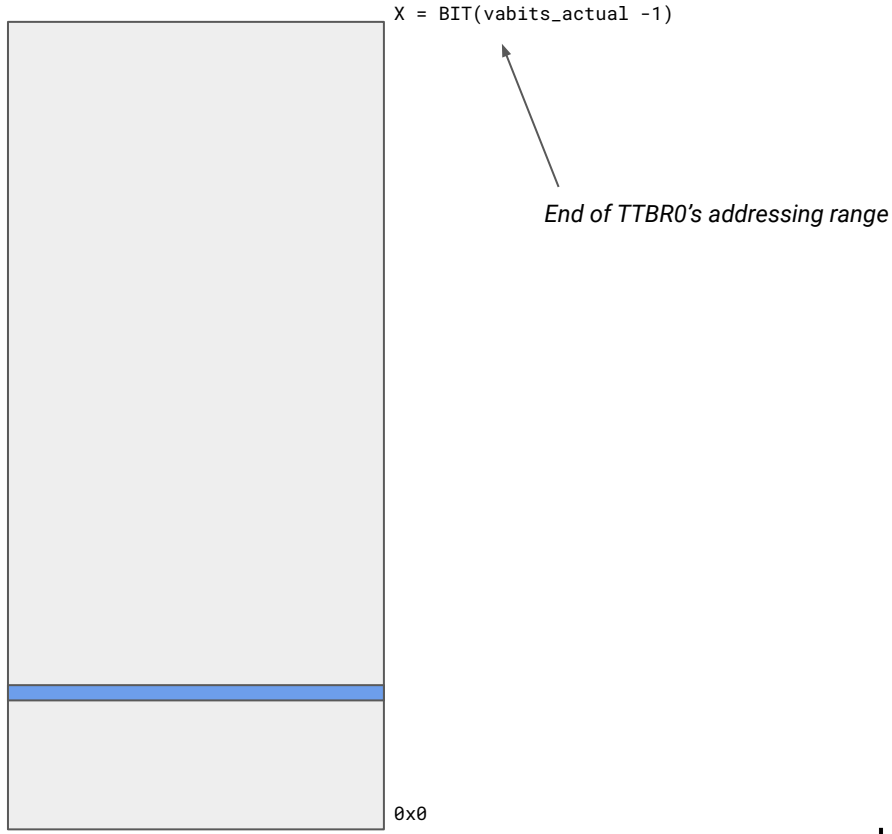
Instruction and data aborts

- When returning to the host because of an instruction or data abort, **ESR** and **HPFAR** are copied in the host's vCPU struct
- KVM converts the GPA in an HVA, and takes a **long-term GUP pin** on the corresponding page to prevent swap and page migration *(more on this later)*
- KVM tops-up a **per-vcpu memcache**, and issues a `__pkvm_guest_map(pfn, gfn)` hypercall
- The hypervisor **tops up its own memcache** using the host-provided memcache (each page goes through a full donation procedure, including ownership checks and such)
- The hypervisor attempts a host-to-guest **donation** for protected guests, or a host-to-guest **share** for non-protected guests, and returns to the host

Guest teardown

- `__pkvm_teardown_shadow()` can be called when there are **no loaded vCPUs** for the guest
- The hypervisor will walk the guest's stage-2 page-table, and mark all the pages owned by the guest as **pending reclaim**
- It also frees the shadow handle and the shadow data-structures
- The host can then issue `__pkvm_host_reclaim_page()` for each page
- The hypervisor will **poison the page** if it belonged to a protected guest, and map it back in the host's stage-2
- Once reclaimed, the host **drops the long-term GUP pin** on the page
- Stage-2 page-table pages and shadow pages are also **reclaimed by the host**

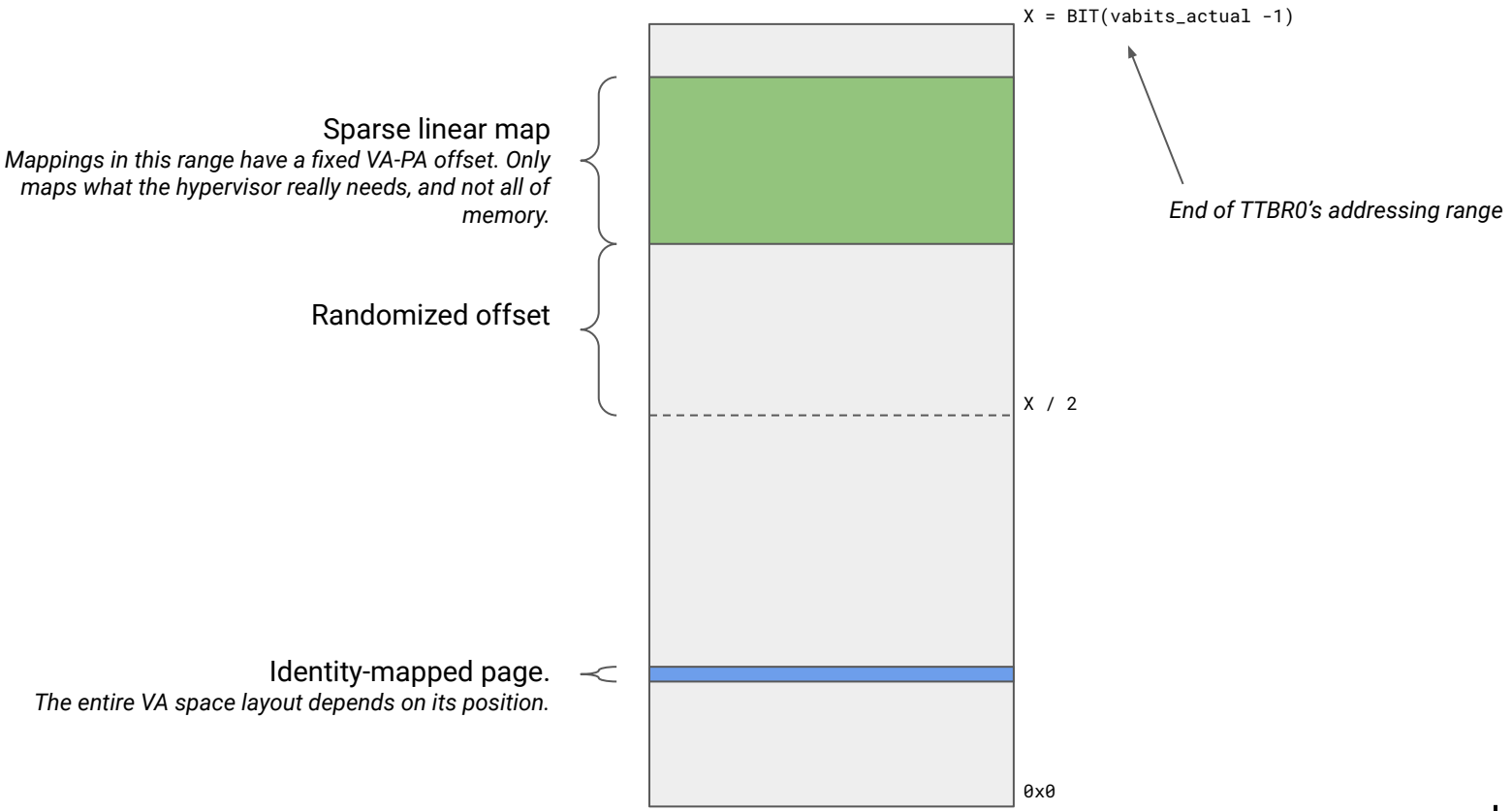
Hypervisor VA Space



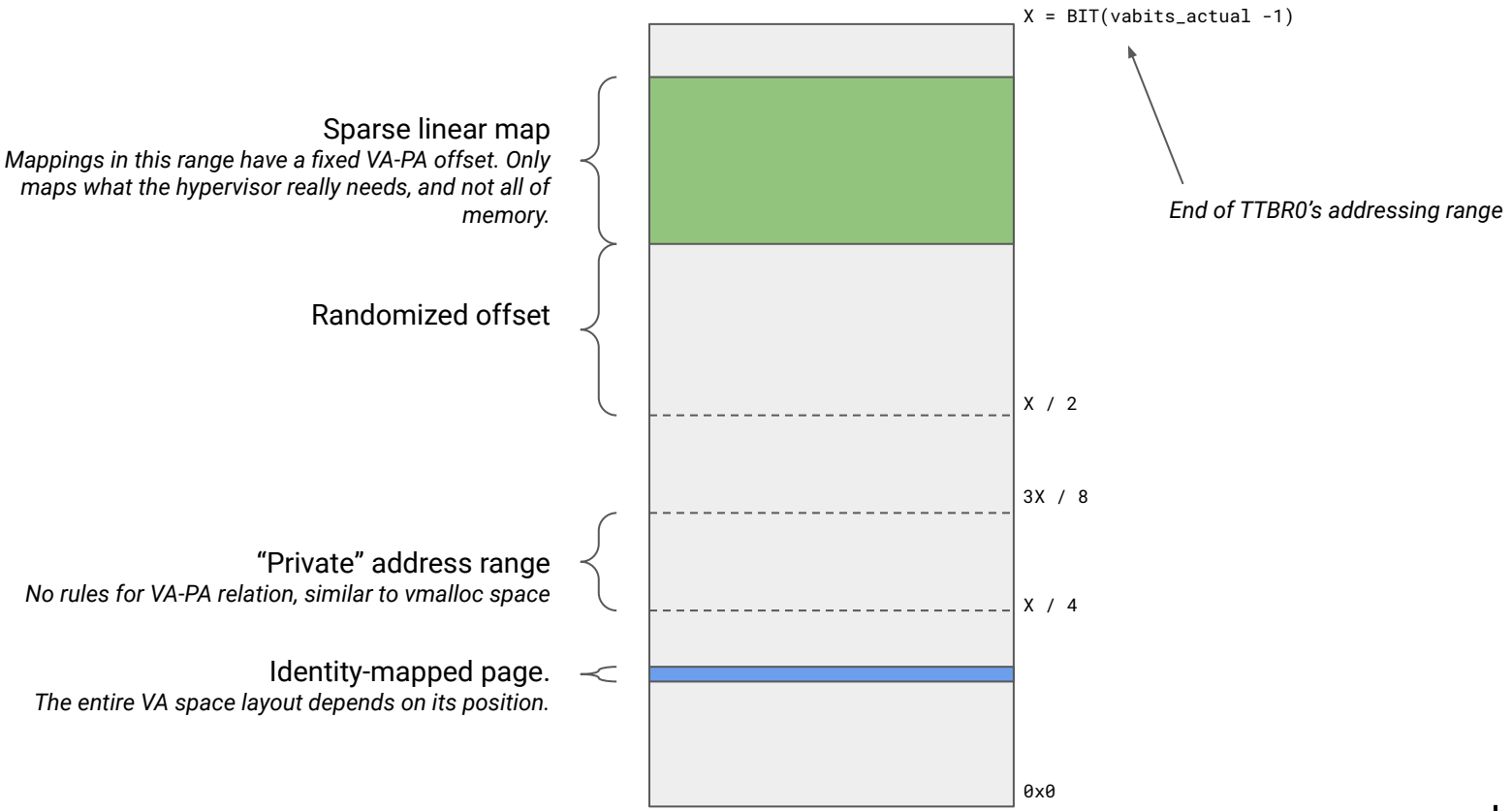
Identity-mapped page.

The entire VA space layout depends on its position.

Hypervisor VA Space

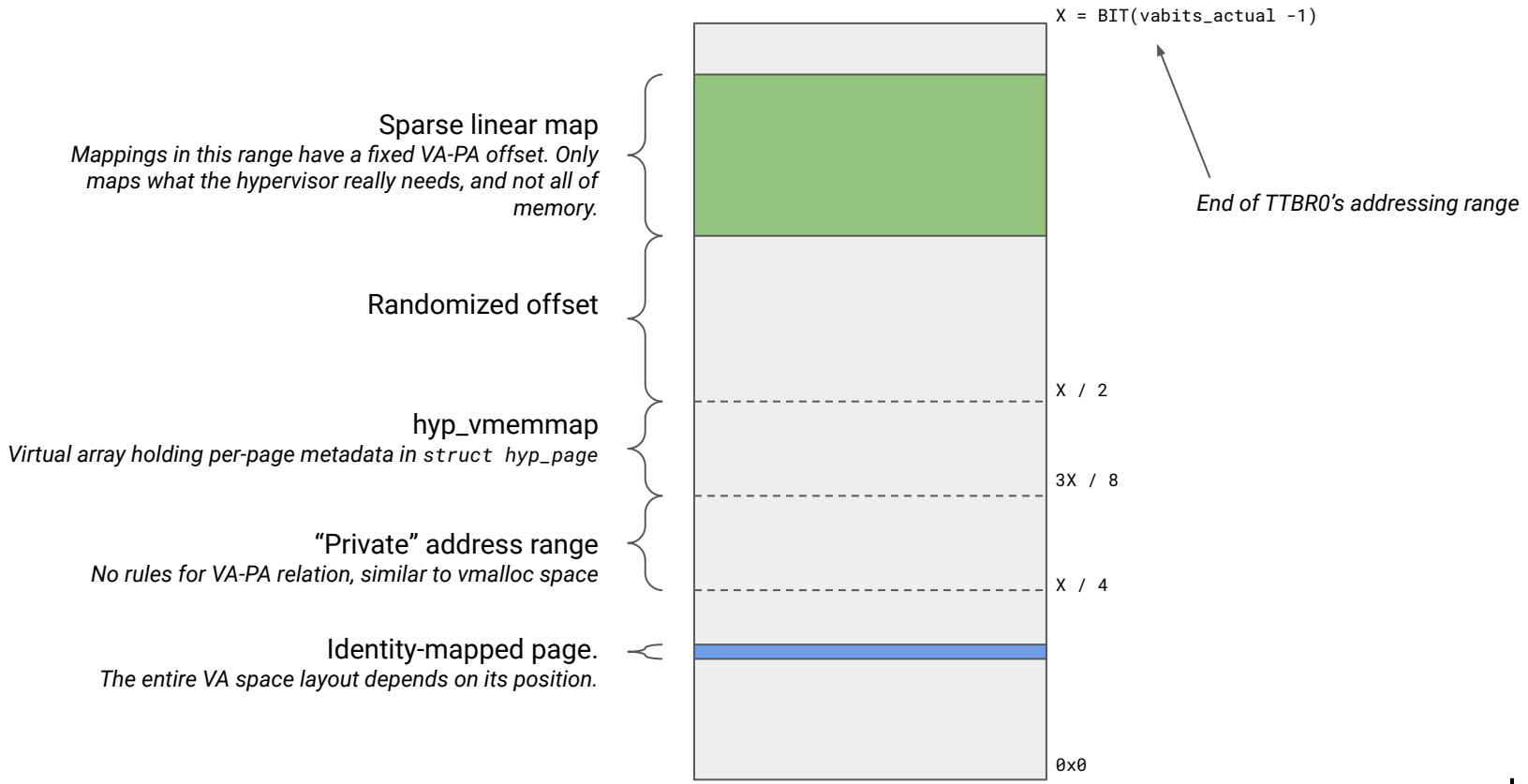


Hypervisor VA Space



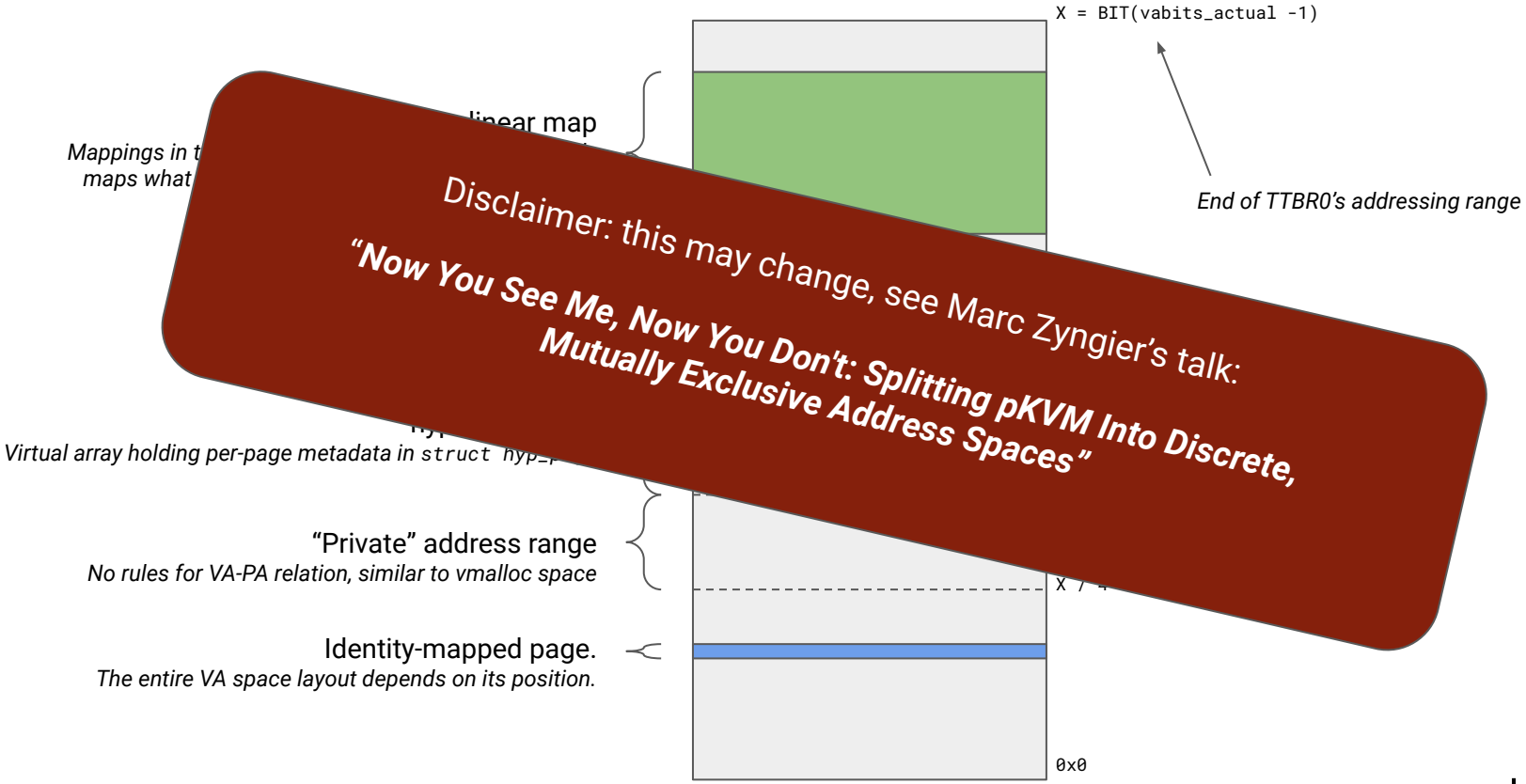
android

Hypervisor VA Space



android

Hypervisor VA Space



android

hyp_vmemmap

```
struct hyp_page {
        u16 refcount;
        u8 order;
        u8 flags;
};
```

Used for:

- *counting PTEs in page-table pages*
- *hyp_{get,put}_page() with buddy allocator*
- *“pinning” a page in shared state*

Used by the EL2 page allocator

Enables memory coallescing of “buddy” pages up to MAX_ORDER

Per-page flags

Currently used in guest teardown path:

- *HOST_PAGE_NEED_POISONING*
- *HOST_PAGE_PENDING_RECLAIM*