KVM/arm64 : Episode V
The Blob Strikes Back

KVM Forum 2023

Oliver Upton <oliver.upton@linux.dev>

Marc Zyngier <maz@kernel.org>

June 14, 2023

Confidential Computing, definition

- Protecting the execution of a given piece of SW
  - Any data that hasn't been explicitly shared must be confidential
  - It must not be possible to infer data from execution artefacts
- SW must be able to verify that it runs on something that offers these guarantees
  - Usually achieved by providing a *measurement* of the whole SW stack
  - ... and the HW platform
  - Use of some *attestation* service
- What is not (usually) provided
  - Availability

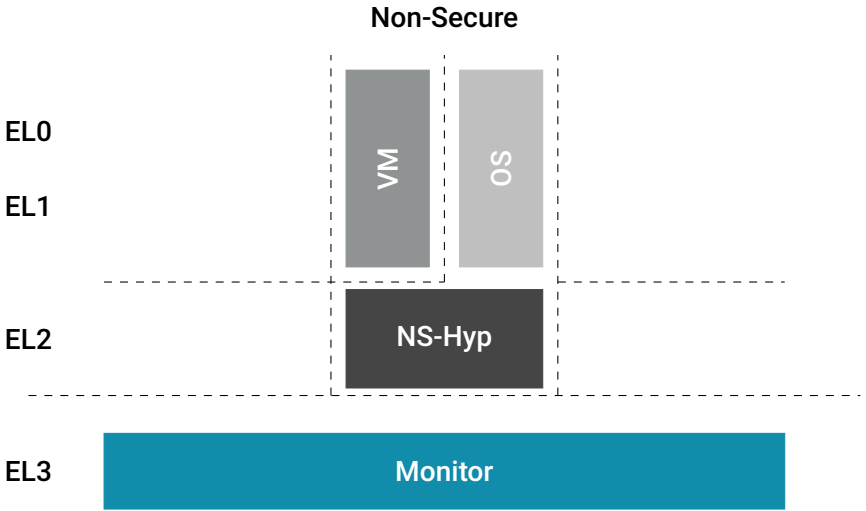
ARM CCA

The *Confidential Computing Architecture* (aka CCA) is made of various building blocks

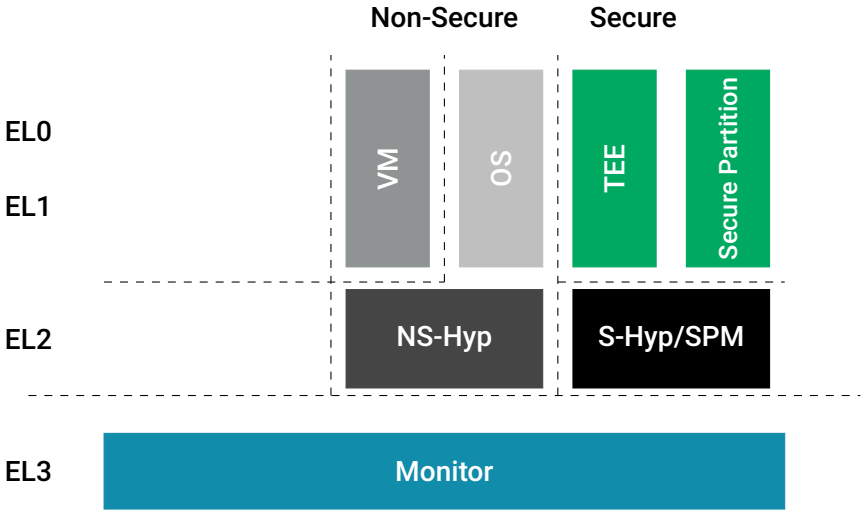
- RME: Real Management Extension
  - Adds a new security domain with the traditional 3 exception levels
  - Parallel to the existing two security domains (Secure and Non-Secure)
- MEC: Memory Encryption
  - Adds per-Realm encryption
- GPC: Granule Protection Check
  - Enforces cross security domain isolation on a 4kB basis
  - Excusively managed at the highest privilege level
- RMM: Realm Management Monitor
  - SW construct acting as a proto-hypervisor for confidential VMs
  - Coordinates with GPC to ensure isolation

RME and RMM are the ones we're interested in...

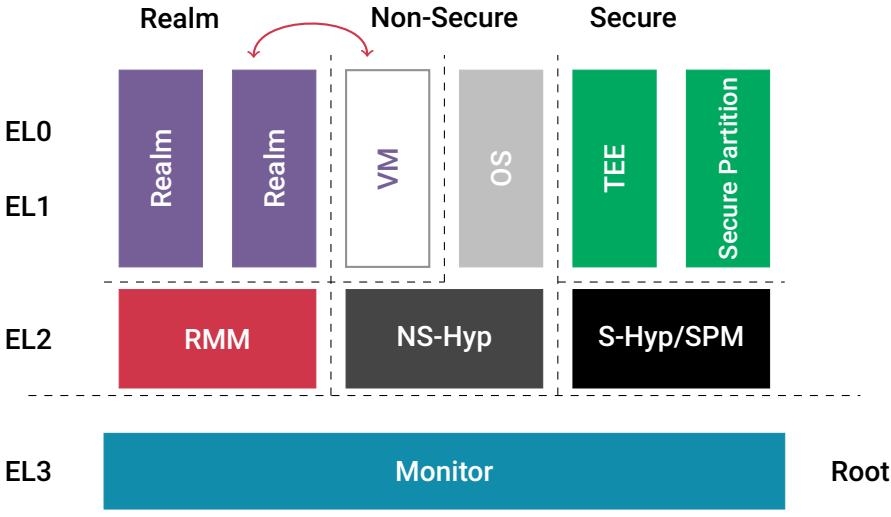
ARM RME



ARM RME



ARM RME



What can RME do for you?

RME provides the basic infrastructure for memory isolation

- Uses a split isolation model
  - Use the GPC to assign a given 4kB block to a given security domain
  - Use standard nested page tables to perform additional isolation in the Realm domain
- Uses a split responsibility model
  - The GPC tables are controlled by Root (aka EL3)
  - The Realm Stage-2 page tables are controlled by R-EL2

The RME SW model

Split responsibilities again

- Firmware running in Root is in charge of configuring the GPC
- The RMM is in charge of
  - the guest state
  - the Stage-2 page table management
  - the handling of most traps
  - the handling of the guest requests (hypercalls)
  - ... basically most of the CPU virtualisation functions
- The Non-Secure hypervisor (KVM, for example) is in charge of
  - Providing the memory for the guest
  - vcpu scheduling
  - Userspace interactions
  - Interrupts (at least for now)

Interaction between RMM and the rest of the stack

Interactions with the NS hypervisor:

- Provides a Realm Management Interface (RMI) which allows Realms to be
  - created
  - provided with memory and payload
  - executed
  - destroyed
- Implemented as SMCs relayed by EL3 to the RMM

Interactions with the guests:

- Provides a Realm Services Interface (RSI) which allows guests to
  - request an attestation report from the RMM
  - manage the sharing of memory with the NS side
  - implement PSCI for the purpose of booting vcpus
- Implemented as SMCs handled by the RMM, and potentially forwarded to NS

The model is sensible enough.
It is actually close enough to what pKVM uses.

Except that...

What's the catch?

Although there is an ARM-provided reference implementation...

- ... the RMM is deployed as a system-specific firmware
- The NS-hypervisor cannot provide its own
- No guarantee over correctness
- No guarantee that it will be updated when bugs are found
- Not even a *method* to update it

The API has been developed as a "one-size-fits-all"

- Or rather *only* for a hypothetical hypervisor
- Defines policies rather than mechanisms
- No access whatsoever to the underlying HW architecture

The whole thing has the flexibility of a concrete wall...

Hypervisor as firmware: what's the problem?

Firmware on ARM has a pretty ugly history, starting with TrustZone

- ARM provide good reference implementations (ATF, for example)...
- ... which are forked, hacked and tweaked by SoC vendors
- This results in a very fragmented ecosystem
  - Incompatible behaviours, *value add* features
  - Power management at EL2 is still a common feature...
- These implementations are hardly ever updated
  - Let alone audited/reviewed
  - `ARM_SMCCC_QUIRK_QCOM_A6` is a personal favorite
  - Validation only occurs with ancient software
  - Remember Java's "Write once, debug everywhere"?
- Firmware on ARM tends to be a *ship and forget* business

Can we trust our VMs to this?

The Platitude of ’Zero Trust’

’Zero Trust’ is not fundamentally incompatible with KVM owning part of the TCB

- Trust is a two-way street
  - From the guest perspective, the NS hypervisor is completely untrusted
  - From the NS hypervisor, we’re handing away our VMs to *the blob*
- The CCA ecosystem has been built around the expectation that the *reference hypervisor* is not independently verifiable
- Of course, this is simply not true for KVM
  - All of KVM’s supposed sins are on full display
  - KVM being open source is the basis of another virtualization TCB (i.e. pKVM)
- Since when did we lose trust in a hypervisor with a proven 10 year track record?
- Surely firmware will do a better job than us...

The mythical “common API”

Why is a rigid API a problem?

While the RMM is supposed to be good enough for *everyone*

- It prevents any sort of HYP/RMM optimisation
- Only the *most common* interfaces are implemented
- Everybody has to marshal to/from the standard interfaces
- Effectively a race to the bottom

What will happen when a critical flaw in the API will require an incompatible change?

- Do we immediately revoke support for non-upgraded implementations?
- Will we even have a say in the fix?

The core issue here is that we're not in charge of our own destiny.

Lessons learned from KVM/pKVM

The boundary between the host and the isolation primitive must be flexible

- Critical for security
  - An inflexible API is a showstopper *when* a security hole is exposed
  - Being able to update the security primitive in lockstep is key
- Critical for performance
  - Different hypervisors have different primitives and behaviours
  - Shoving the diversity of the ecosystem into the *one true API* looks wrong
  - Can only result in poor performance and difficult maintenance
- Critical for innovation
  - We're stuck with what the reference hypervisor has deemed useful
  - What if we want userspace-only realms? We can't
  - What if we want Nested Virtualisation inside a realm? We can't

All of this seems to have been ignored...

Case study: the journey of an EL1 timer interrupt

The RMM does near nothing in terms of GIC emulation

- Manages the GICV CPU interface, but:
  - NS hypervisor still responsible for driving it (i.e. interrupt injection)
  - Global (distributor) and local (redistributor) state lives in NS hypervisor
  - RMM doesn't have a view on the global GIC state
- While the RMI interface is correct, it might not be the most performant design
  - *Any* physical interrupt requires a full exit out to the NS hypervisor
  - Same goes for physical interrupts destined for the guest (EL1 timer, for example)

Case study: the journey of an EL1 timer interrupt

Wouldn't it be better if we had a shared-memory interface for GIC state?

- RMM could theoretically handle some virtual interrupt injection
  - Guest timer interrupts are a very obvious point of optimization
  - Could reasonably be extended to other interrupt sources assigned to the guest
- Requires a more complicated interface, harder to get it right
  - In the current model it requires an architected interface to work with a *reference hypervisor*
  - A flexible API with the RMM would let us improve *when* we get it wrong

Ecosystem perspective

Direction of travel

It is interesting to see how (and where) the embedded world is moving

- pKVM is an opportunity to move “secure” services into VMs
  - Breaking the confidentiality/privilege dependency
  - A change Android sorely needs
- It does so *without* binary blobs that cannot be replaced
  - The payloads are confidential
  - The hypervisor is for everyone to see, run and improve
- It wouldn’t be hard to move the NS-EL2 side of pKVM to R-EL2
  - The HW architecture actually lends itself to it very well
  - pKVM would effectively have its own, tightly coupled RMM

And yet, the current CCA architecture actively prevents such a move.

At odds with the open source ecosystem

Onus is on KVM to align RMM API with the rest of the open source ecosystem

- Make no mistake: CCA seeks *reuse/abuse* the pre-existing KVM ecosystem
  - KVM becomes nothing more than a userspace <-> firmware proxy
  - Requires KVM to make CCA compatible with present and future KVM features
- Vendored firmware will invariably require vendor-specific workarounds
- Introduces toil for developers largely unconcerned with CCA
  - Is it reasonable to expect developers to support features across KVM/pKVM/CCA?
  - Should upstream tolerate features *incompatible* with the RMM API?
- Divides the already finite attention of virtualization-minded open source developers
- Assumes that the hypervisor is, by definition, unable to deprivilege itself

OK, enough ranting

What do we need (and when do we want it)?

To move forward and make CCA a first-class KVM implementation, we need:

- An *architected, secure way* to deploy a RMM at boot-time
  - Architected so that we don't have to deal with 36 different methods
  - Secure, so that EL3/Root can *measure and attest* of the validity of the RMM to a guest
- A measurement mechanism is independent of the RMM implementation
  - Should exist as an architected contract between RMM and guest
- A process that leverage the existing Secure Boot *infrastructures*
  - Because different OSs have different requirements (UEFI-based boot vs Android, for example)

And we need this sooner rather than later.

Is the whole CCA saga wasted effort?

Do we need to throw away everything? Absolutely not!

- The HW architecture is sensible
  - Provides the required isolation primitives
  - Even the EL3/Root control of the GPC is OK, as long as it is performant enough
- Most of the RMM's RSI is probably worth keeping
  - Nothing controversial there
  - It'd be even better if RSI and pKVM's hypercall interface could converge
- Even some of ARM's RMM reference implementation could be reused
  - A common RMM library would be very useful
  - The API is what we object to

What do other architectures do?

We've been ranting about ARM. Are the other architectures any better?

- x86 has it relatively easy
  - Only two vendors
  - Strong control over the "API" (ucode FTW!)
  - Suffers from being very inflexible ("*we know better than you do*")
  - Well established upgrade path
- RISC-V looks like ARM, at least with CoVE with the host in HS mode
  - TSM firmware (part of SBI) running in M-mode
  - Another variant with the firmware running in HS mode
  - Pre-baked API (COV{I,G}, COVH)
  - In any case, yet another fragmentation risk
- S/390: please tell us about it!

Is it time for lunch yet?

Conclusion

Here's a plea to ARM:

- CCA is one of the major ARMv9 features
- It has the potential to redefine how workloads are isolated on ARM systems
- Ensuring that the use of the architecture isn't limited is essential
- KVM has been a key enabler for the success of ARM in the data-centre
- We want to ensure that this success continues
- We want to keep innovating in this field

Please give us the tools Open Source hypervisors need to meet this ambition.

Thank You!