



Secure Initialization of TEEs

when secure boot falls short...

Cristofaro Mune (@pulsoid)

Eloi Sanfelix (@esanfelix)

EuskalHack 2017

Who?

- **Cristofaro Mune**

- Embedded Security Consultant (Independent)
- Keywords: TEEs, IoT, Embedded SW & HW, Fault Injection
- Previous work: WBC, IoT, Embedded Exploitation, Mobile



- **Eloi Sanfelix**

- Principal Security Analyst @Riscure
- Keywords: Software security, TEE, RE, Exploiting, SCA/FI, CTF
- Previous work: WBC, DRM, PayTV, Smart Cards

riscure

Challenge your security

What and why...

- TEEs Increasingly relevant in security solutions

...Basically everywhere

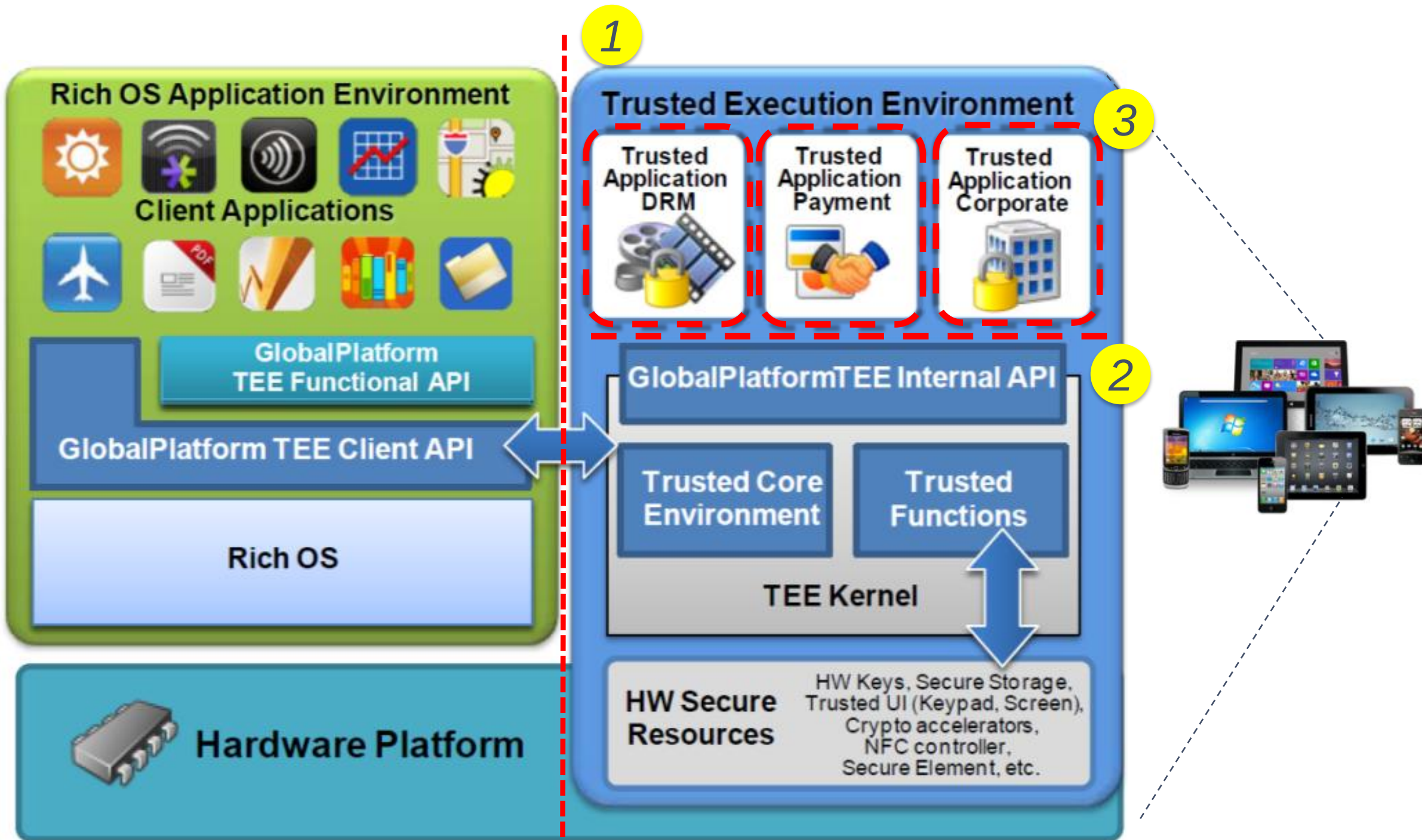
- Research:
 - Interesting but limited in amount and scope
- Lack of a generic TEE security modeling
 - Components and Mechanisms
 - Attack surfaces
 - Attack vectors

TEEs: Fundamentals

Trusted Execution Environment (TEE)

- Aimed at providing a secure environment for execution of security critical tasks:
 - Payment applications
 - DRM applications
 - ...
- Separated from Rich execution environment (REE)
 - Non-secure, untrusted environment
- Support for Trusted Application (TAs):
 - Separated from each other
 - Typically implementing one single use case

System overview



TEE Critical items

1. TEE separations:

- 1. Separation from the Rich Execution Environment (REE)**
2. Separation between TAs and the TEE OS
3. Separation between TAs

We focus on this...

...but concepts also apply to these.

Strong cooperation between HW & SW

HW & SW roles

Hardware protecting

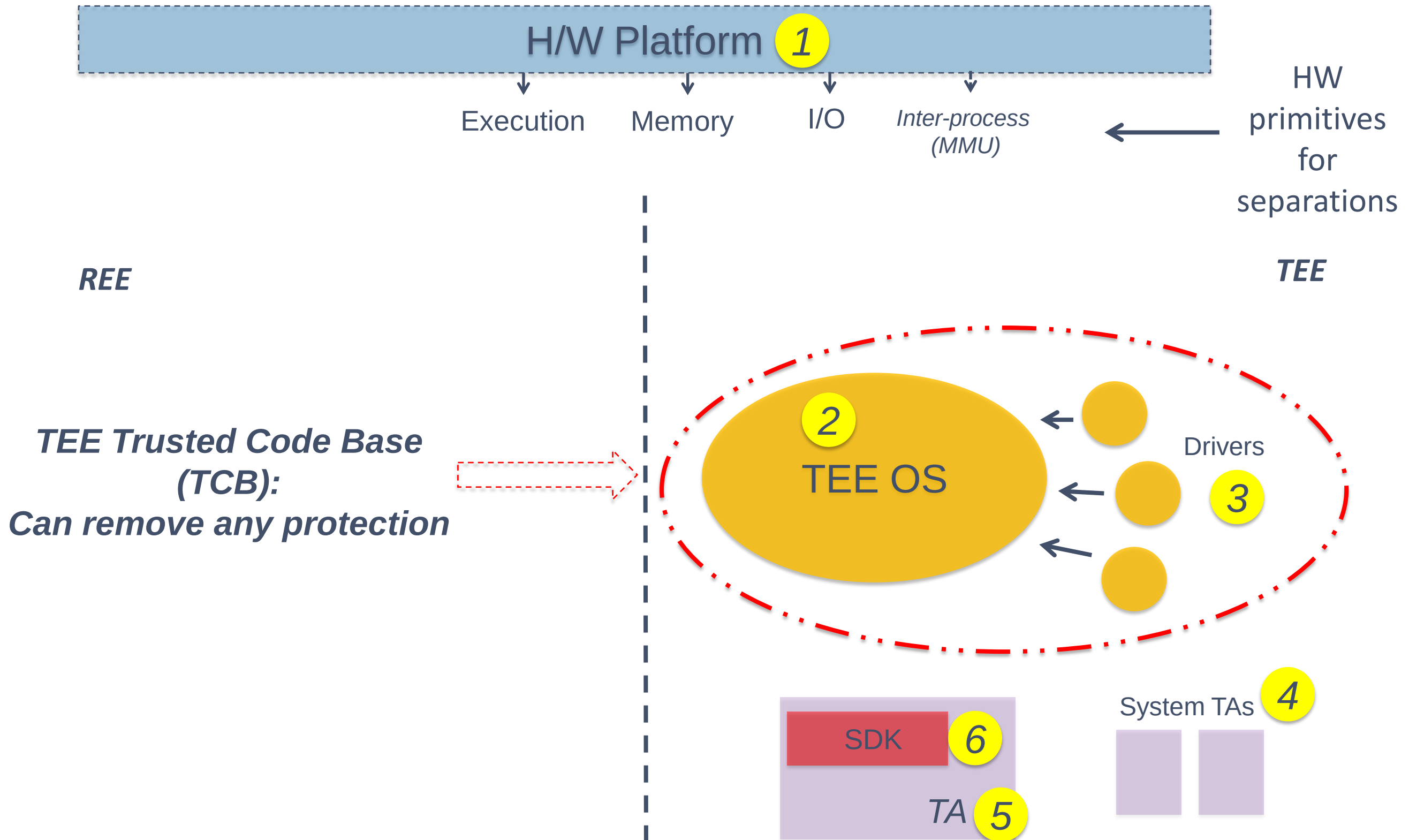
Software

+

Software protecting

secrets

A TEE reference frame (runtime)



ARM TrustZone

Example SoC: CPU

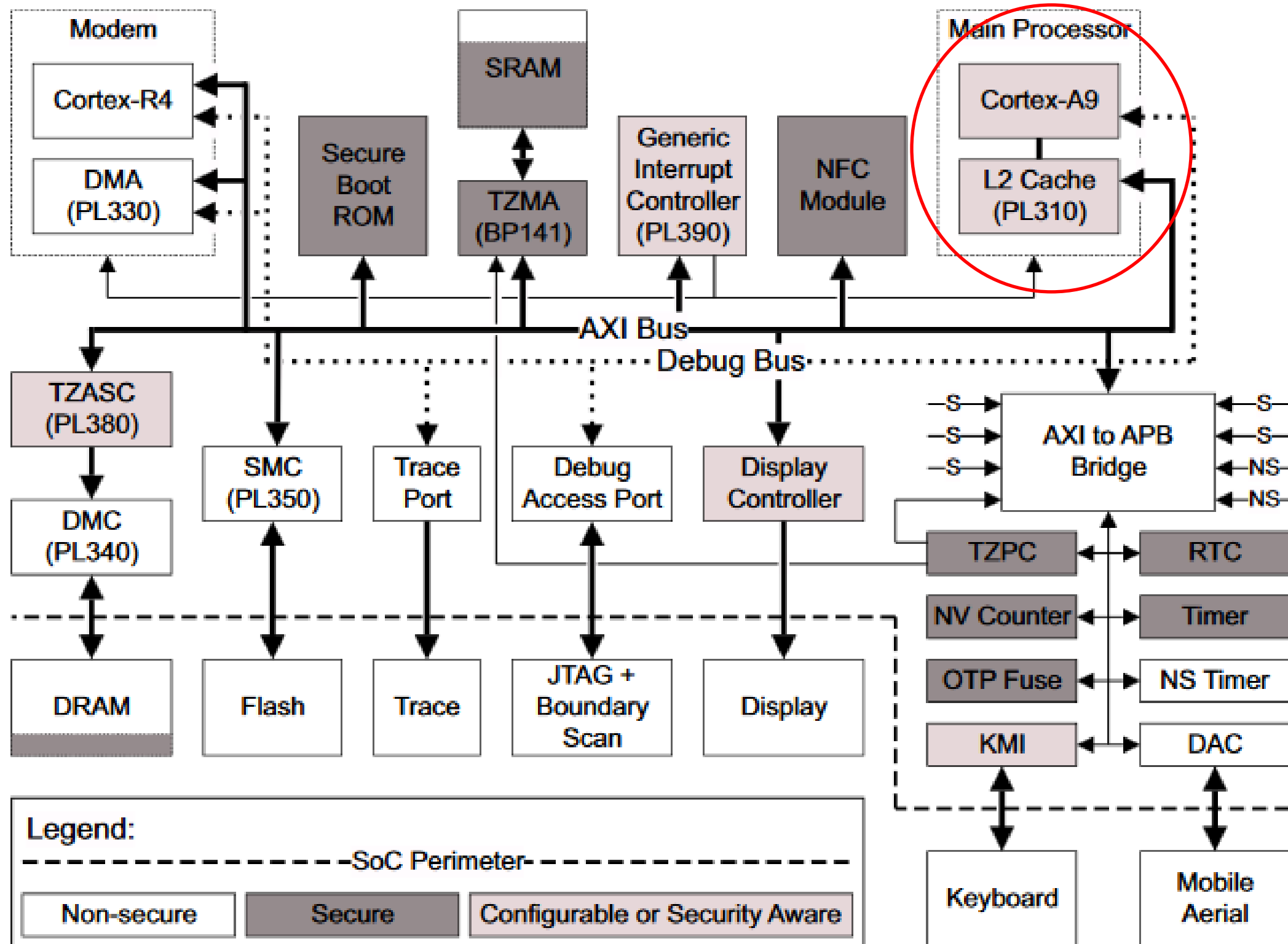
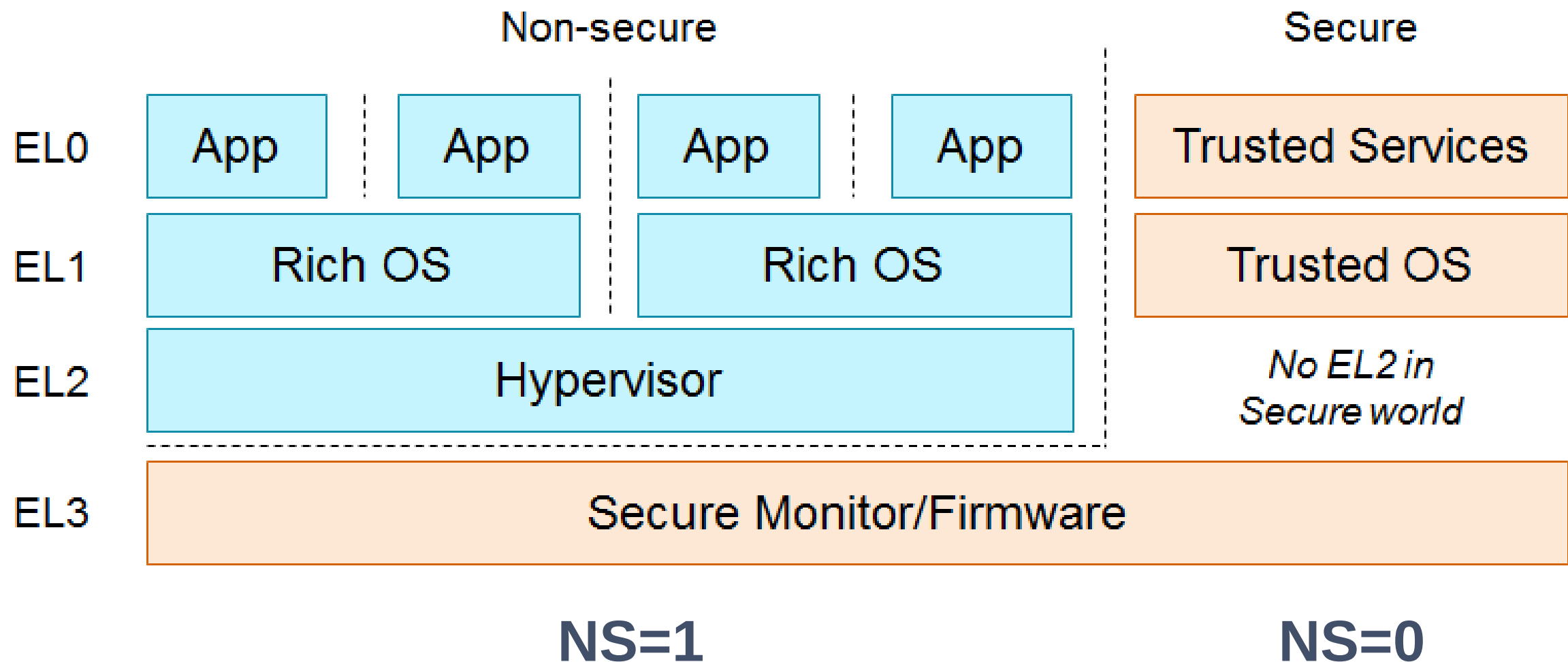
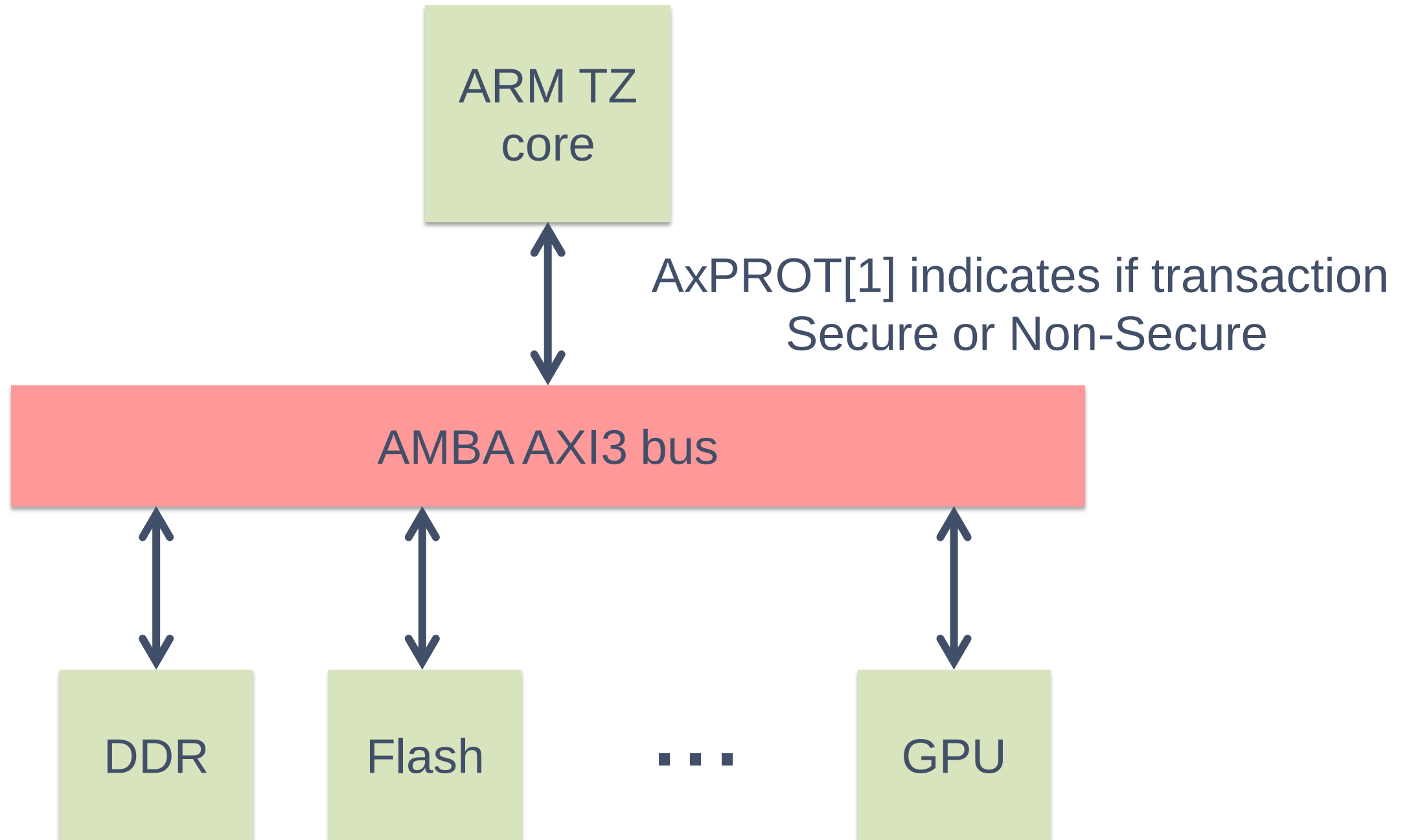


Figure 6-1 : The Gadget2008 SoC design

CPU Security State



Security State propagation



How is AxPROT[1] determined?

- All AXI slaves are memory mapped
 - Including DDR, HW registers, etc.
 - Page Table Entries include an NS-bit
- AxPROT[1] depends on CPU and PTE NS bits

CPU NS	PTE NS	AxPROT[1]
0	0	0
0	1	1
1	X	1

Example SoC: protection enforcement

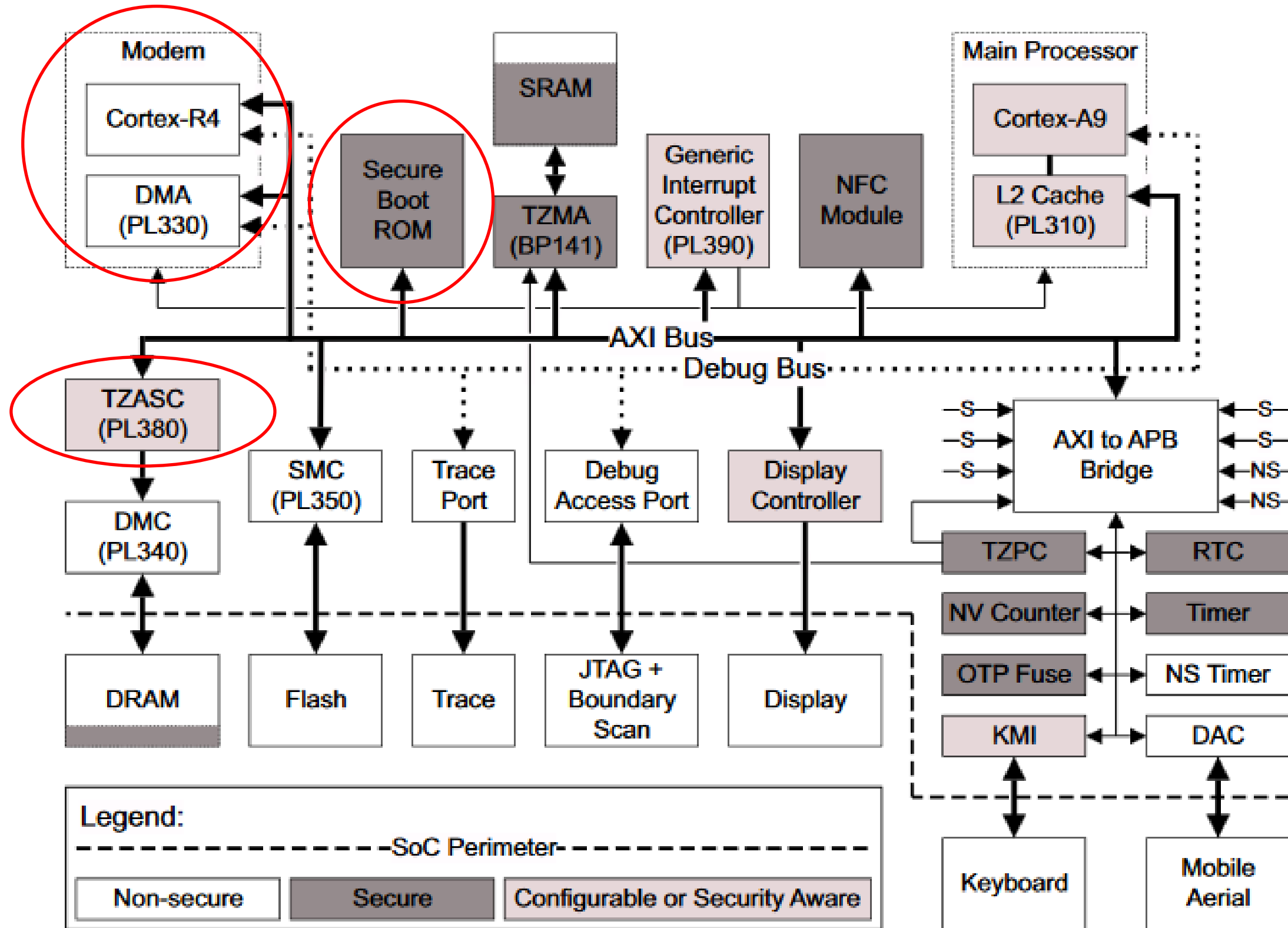
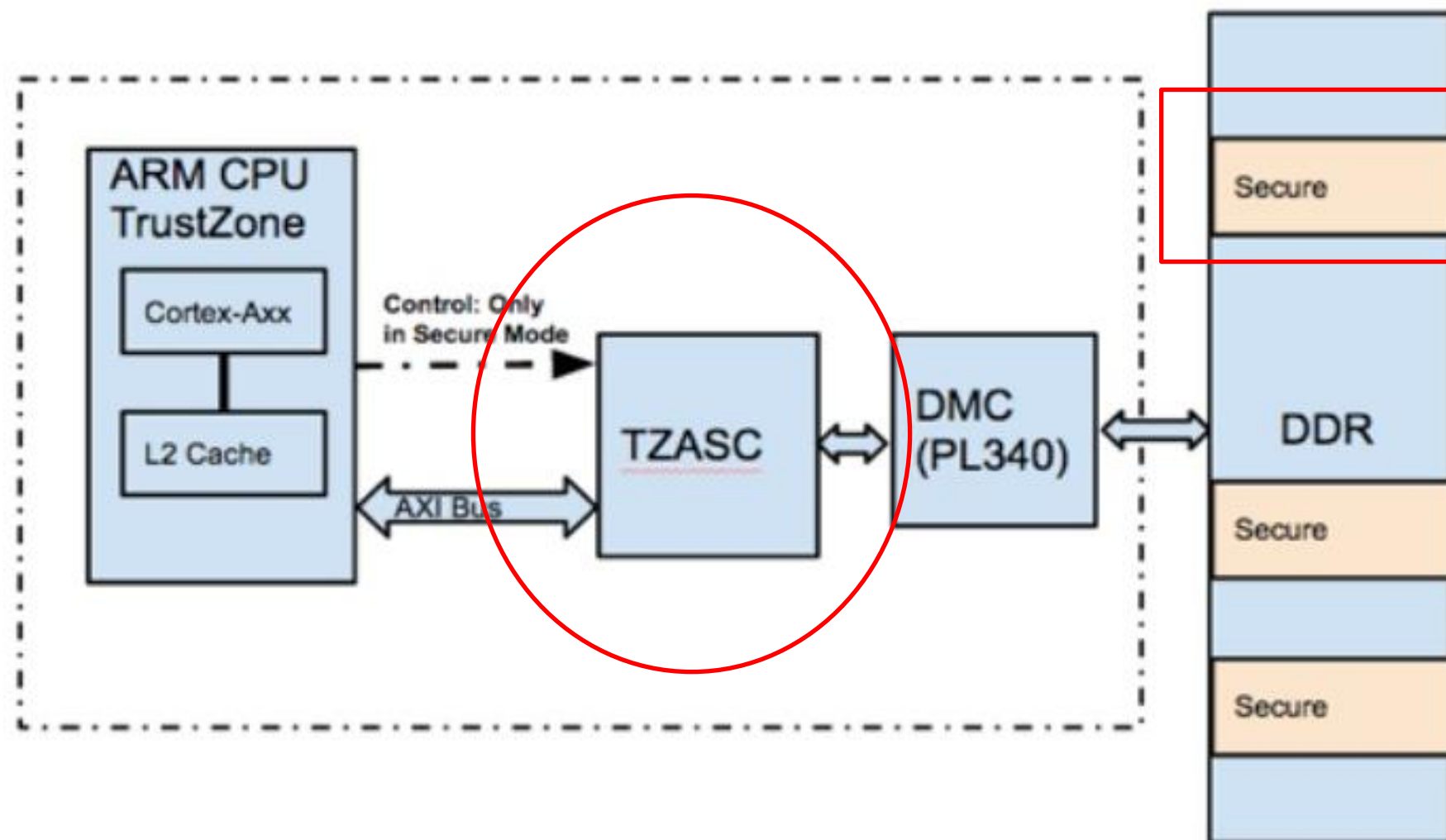


Figure 6-1 : The Gadget2008 SoC design

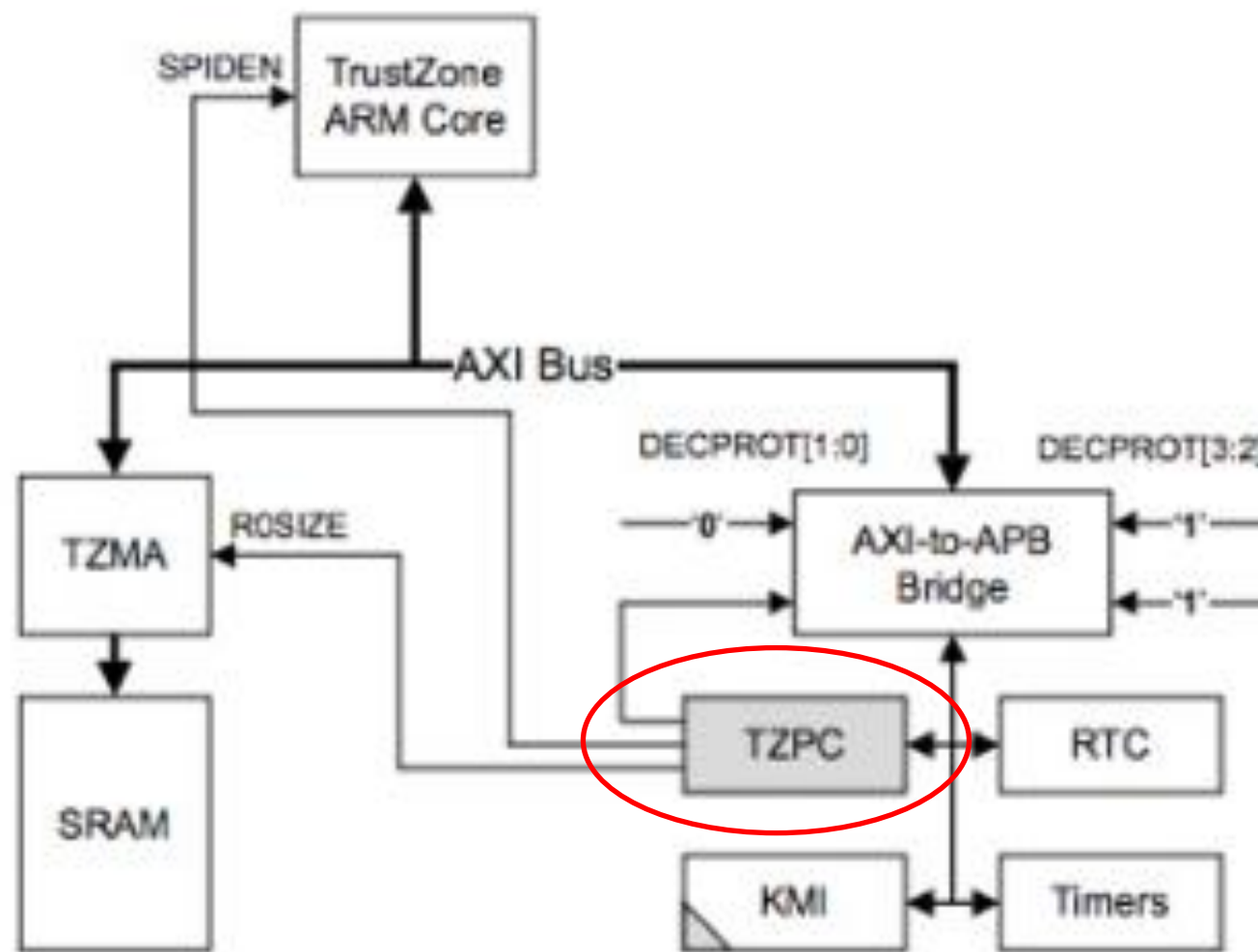
Example: Protecting DDR memory

Hardware: TrustZone Address Space Controller (TZASC)



Example: Protecting peripherals

Hardware: TrustZone Protection Controller (TZPC)

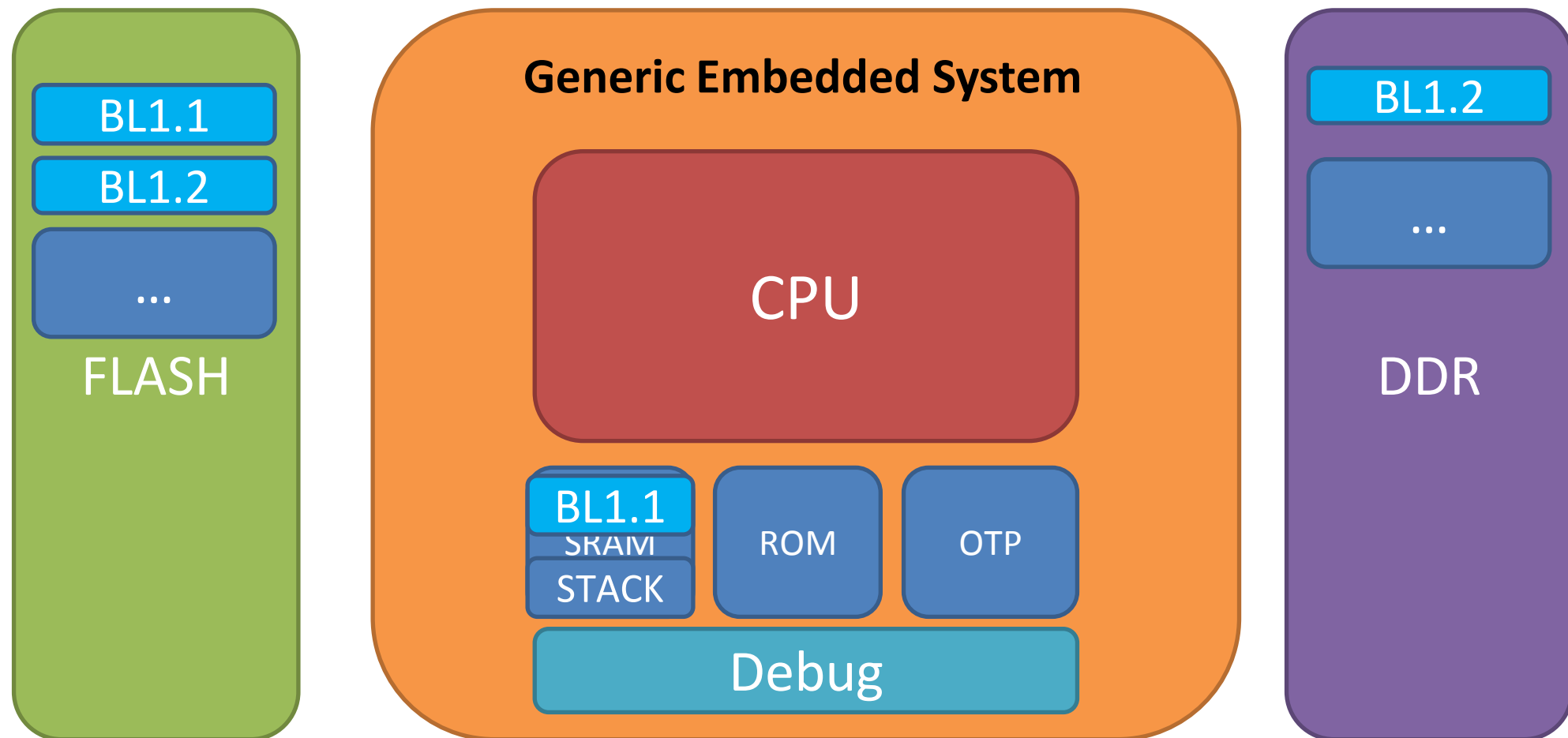


What about other slaves?

- AXI slaves in charge of enforcing transaction security
- Can be done with:
 - Controllers (TZASC, TZPC, etc)
 - Hardcoded logic in bus matrix
- *Controllers **MUST** be configured by SW*

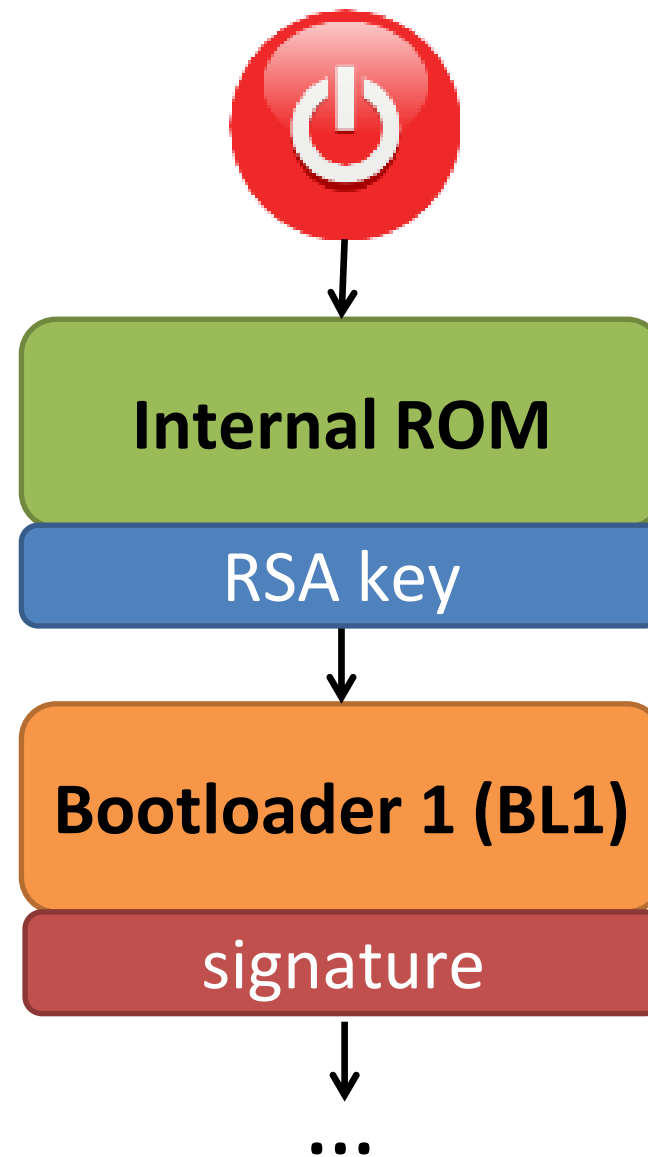
Secure Boot

Why Secure Boot?



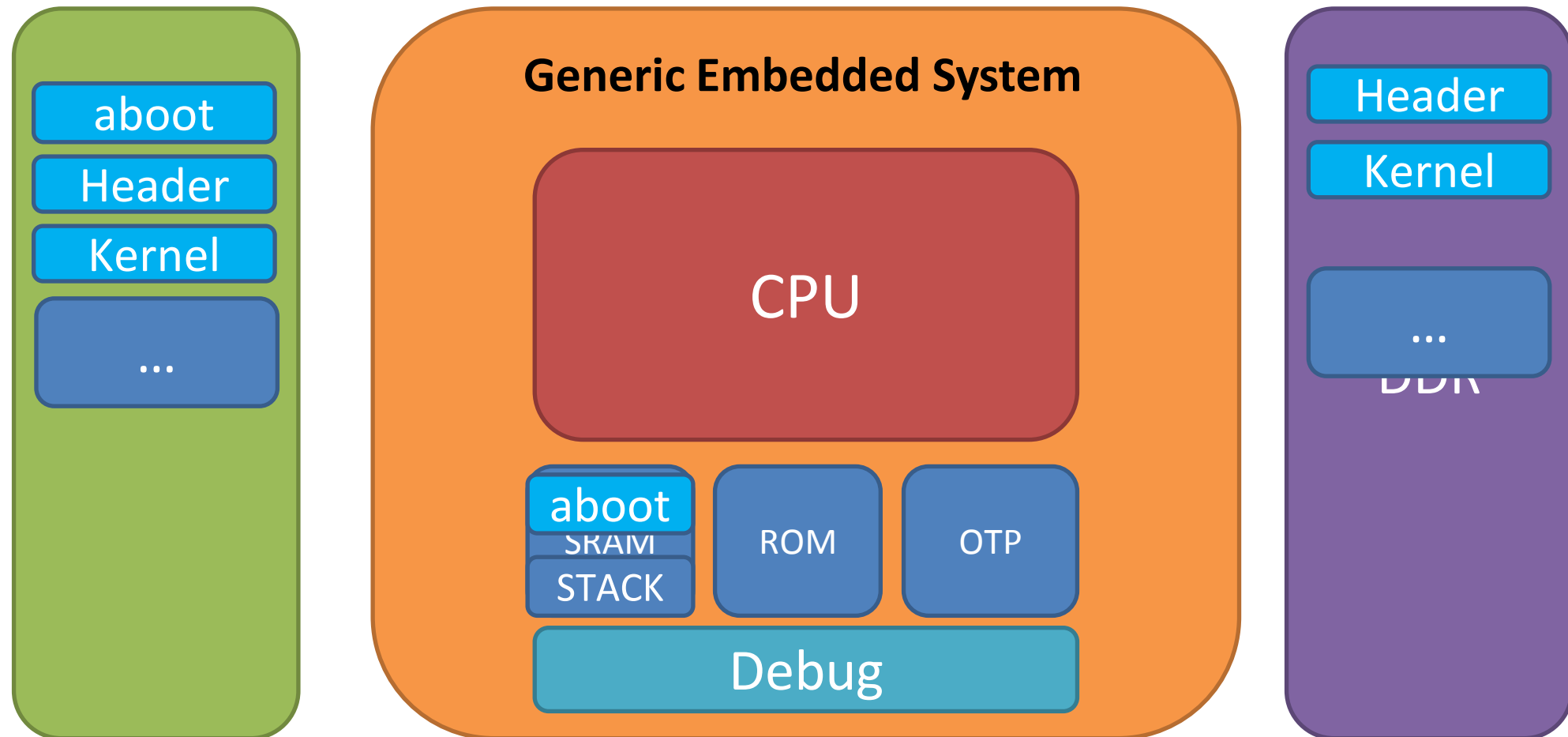
- Integrity and confidentiality of flash contents not assured
 - TEE security is not established!
- Secure Boot provides this assurance

Typical Secure Boot implementation



- Assures integrity (and confidentiality) of flash contents
- Root of trust composed of immutable code and data

SB vulnerability: Samsung Galaxy S4



1. *about* copies header, then kernel
2. Signature is verified and kernel booted if OK.

Any problems?

```
hdr = (struct boot_img_hdr *)buf;

image_addr = target_get_scratch_address();
kernel_actual = ROUND_TO_PAGE(hdr->kernel_size, page_mask);
ramdisk_actual = ROUND_TO_PAGE(hdr->ramdisk_size, page_mask) + 0x200;
imagesize_actual = (page_size + kernel_actual + ramdisk_actual);

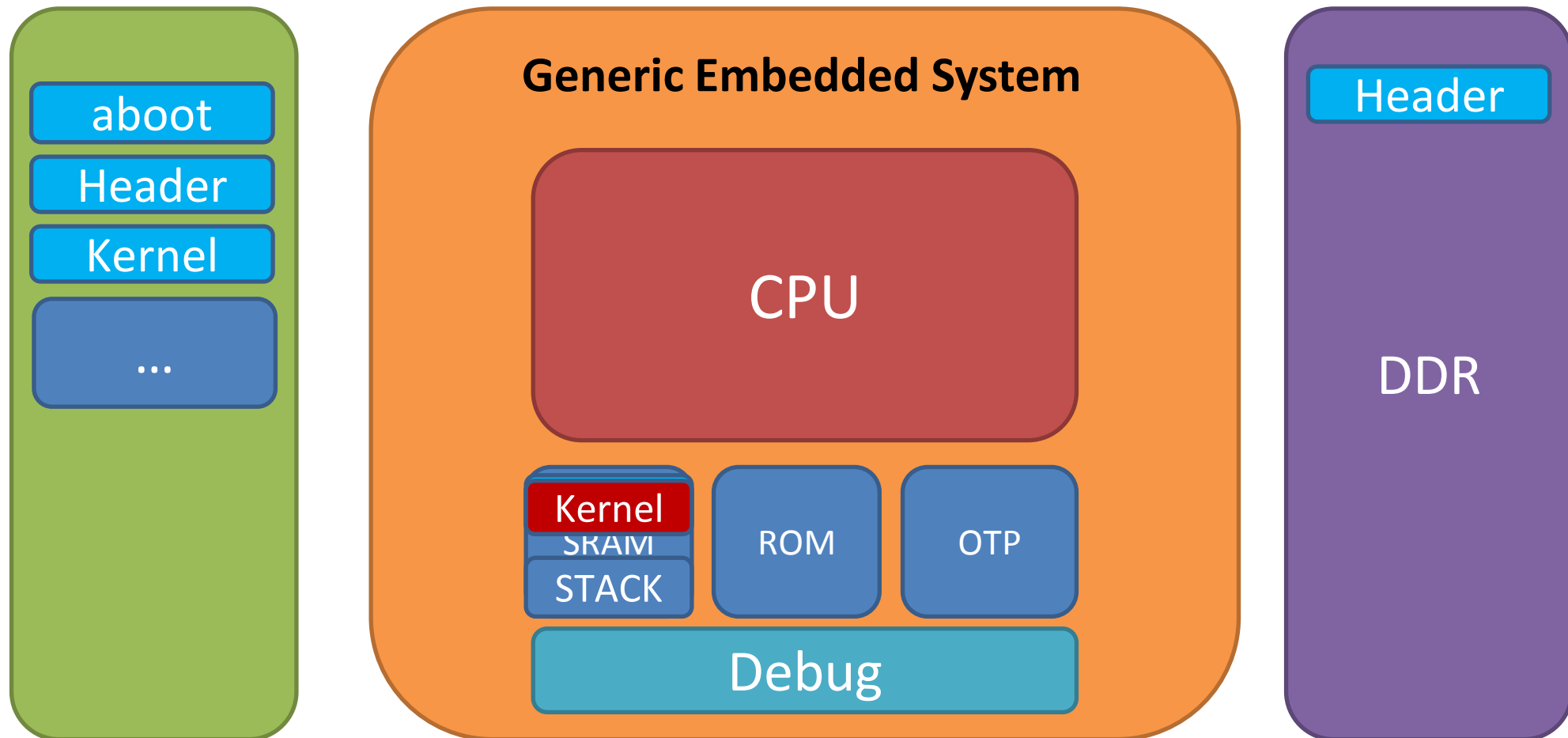
memcpy(image_addr, hdr, page_size);

offset = page_size;

/* Load kernel */
if (mmc_read(ptn + offset, (void *)hdr->kernel_addr, kernel_actual)) {
    dprintf(CRITICAL, "ERROR: Cannot read kernel image\n");
    return -1;
}
```

Untrusted → Arbitrary memory corruption


So what?



- *aboot* smashes its own code with attacker-supplied code!
- Alternatively, attacker could target return address on stack

SB vulnerability: AMLogic S905 SoC

```
int auth_image(aml_img_header_t *img){
    validate_header(img); // checks on magic value & header length
    hash = hash_sha256(img); // hash whole image except signature
    if(img->sig_type == RSA) {
        return check_rsa_signature(img, hash)
    }else{
        return memcmp(hash, (char*)img + (img->sig_offset));
    }
}
```



Untrusted data used to determine whether signature check is enabled!

Beyond Secure Boot

- Secure Boot makes sure code is authentic
 - You still need to set up the REE and TEE!
- In particular:
 - Initialize separations (TZASC, TZPC, ...)
 - Load TEE OS into Secure World
 - Initialize other SoC components

***The TEE needs to be
securely initialized before
running any REE code!***

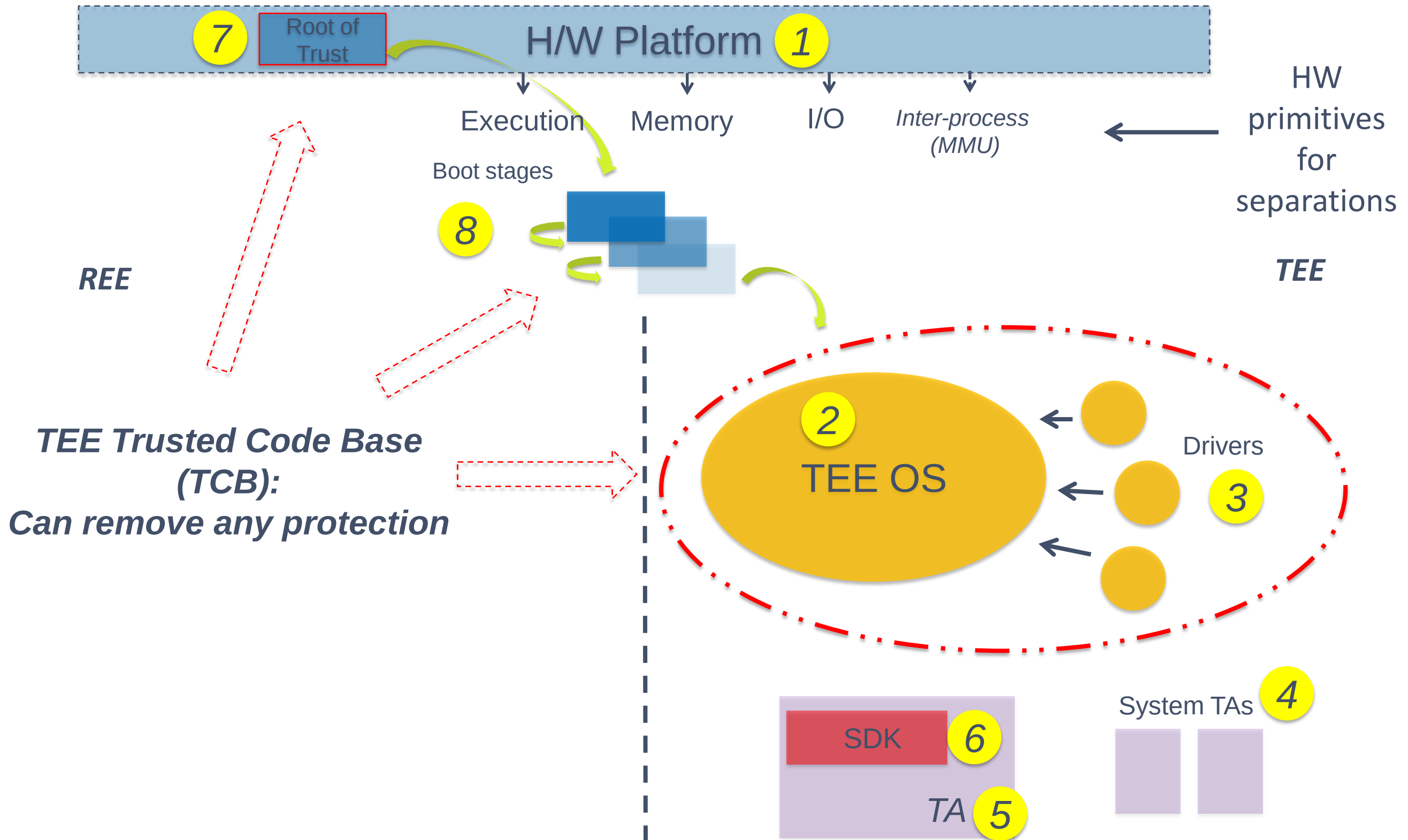
“Time”:

TEE initialization

TEE initialization

- TEE initialization is based on Secure Boot.
- *TEE initialization must also **protect, load, verify, initialize and configure** the TEE.*
- *Then **demote** to REE.*

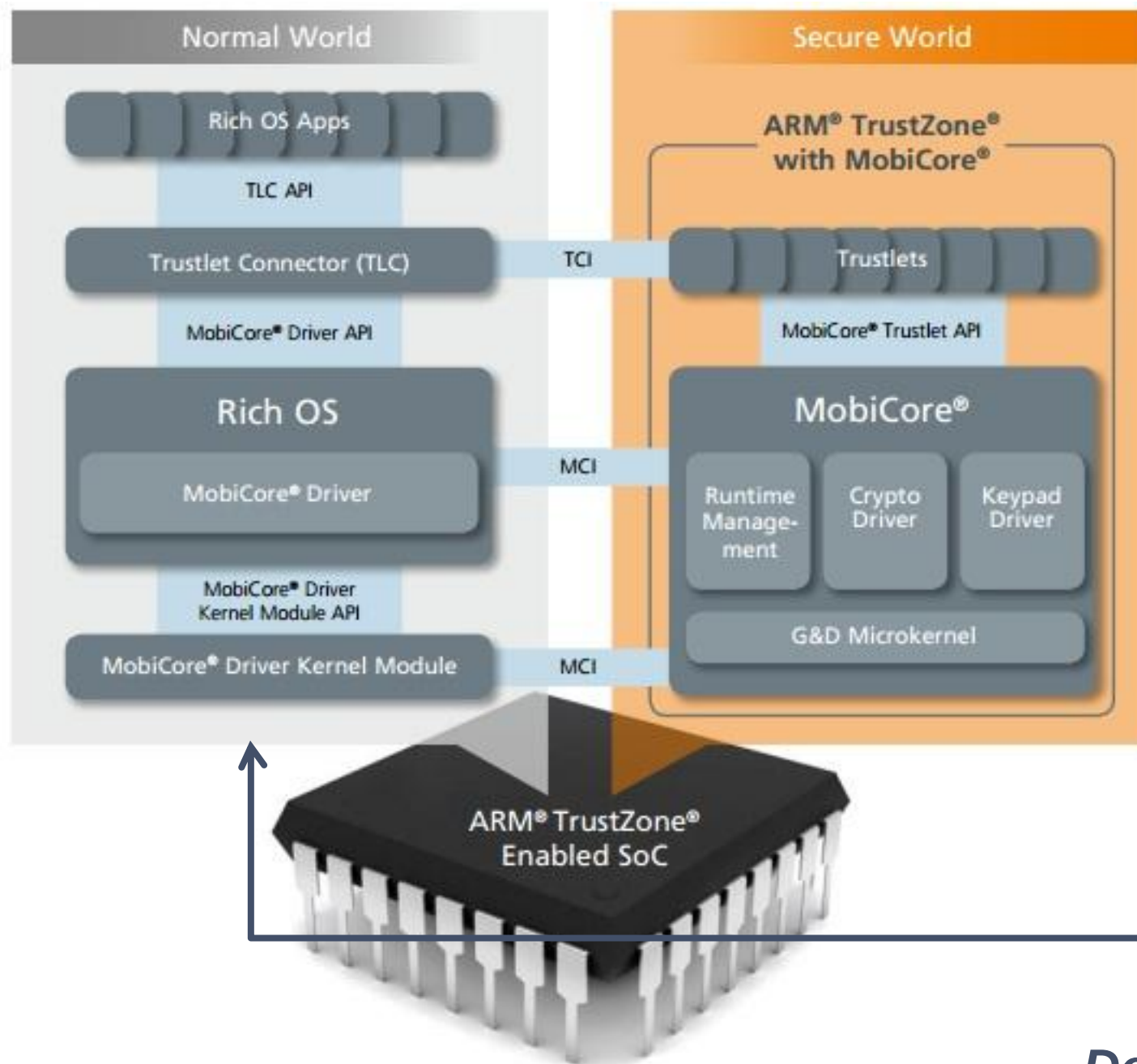
A TEE reference frame (full)



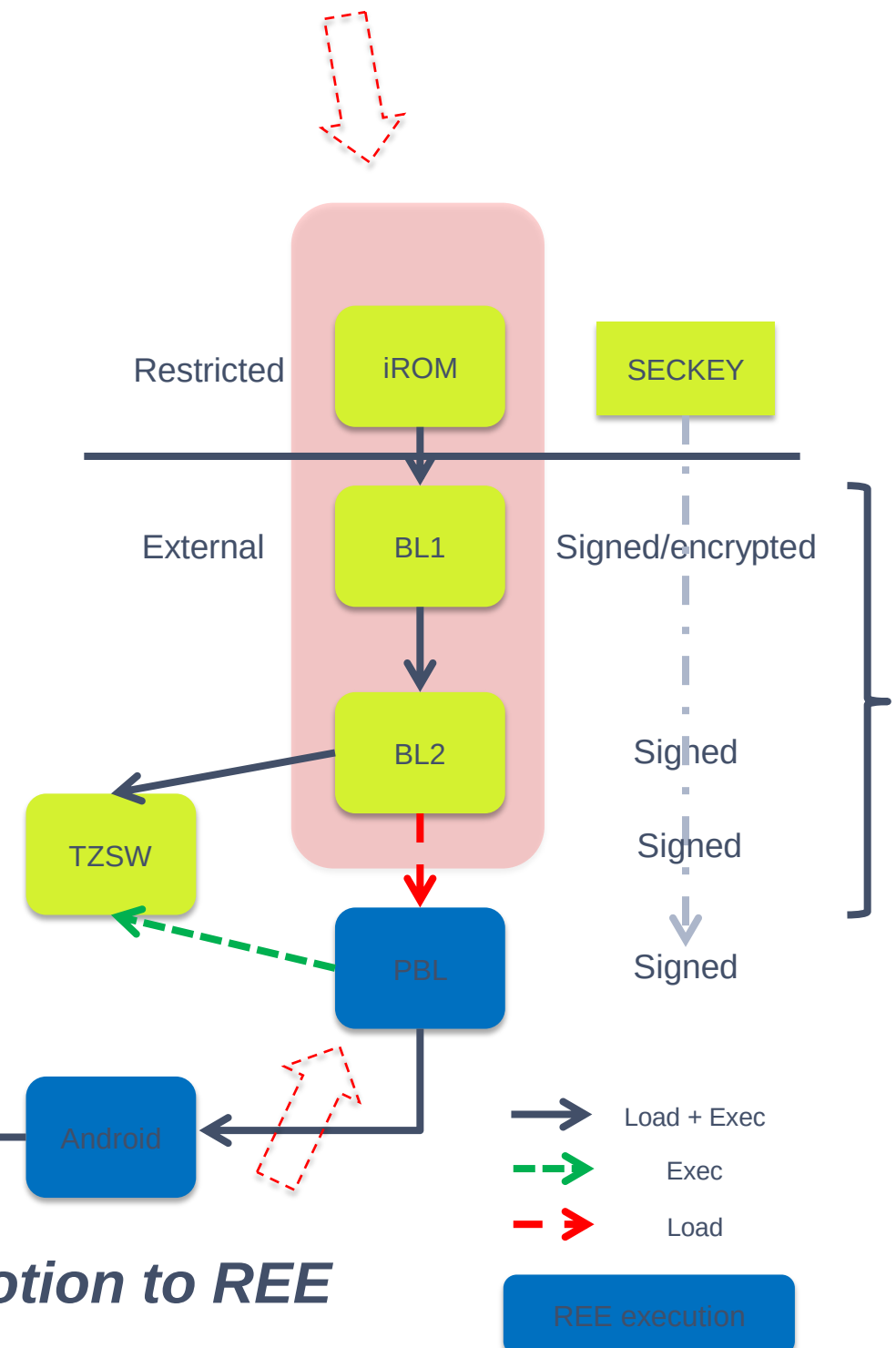
Some definitions

- ***Demotion point:***
 - The point (in time & code) in a boot process, where **ALL** the privileges for configuring a TEE are given up
 - ...and REE is started.
- ***Critical path(s):***
 - The set of all the code paths that can be executed before the *Demotion point*
 - Parts of the TEE attack surface

How it works: Old Samsung phone



Critical paths



Demotion to REE

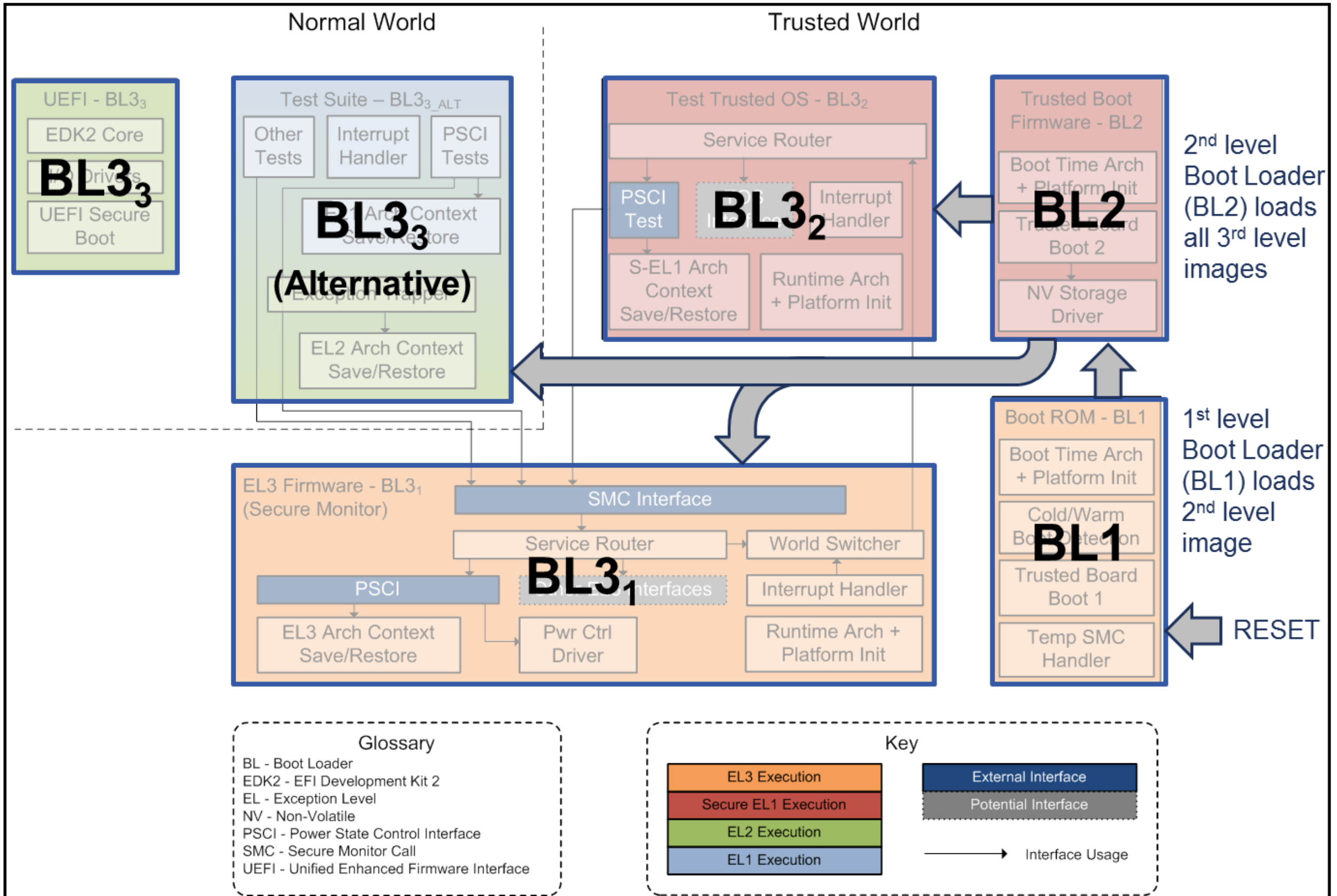
Just “Secure Boot”?

- The following must be executed before Demotion point
- For each TEE-related boot stage:
 - Identify **WHERE** to load the stage in memory
 - **Protect** memory from REE
 - E.G. configure TZASC
 - **Load and Verify.**
 - *Run any stage initialization code*
- **Configure (...more to come...)**
 - *Other IPs*
 - *Other Protection Controllers*

ARM Trusted Firmware

- Reference implementation for trusted TEE initialization
 - ARMv8-A architecture
 - ATF v1.3 now released
 - Security improvements over v1.2
- **Customizations** needed:
 - Highly dependent on memory layout (and design)
 - Examples:
 - Configuration of TZASC and TZPC
 - ...or equivalent controllers
 - Initialization routines for BL31 and BL32

Example: ARM Trusted Firmware



Range checks

- One of TEE security foundations
 - Is it Secure or Non-Secure Memory?

How difficult can it be?

Real world example

[tzbsp_oem_access_item, address validation]

```
#define IS_TZ_MEMORY(x) (x >= 0x2A000000 && x < 0x2B000000)

int tzbsp_oem_access_item(int write_flag, int item_id, void * addr, int len) {
    if (!is_svc_enabled(26)) {
        return -4;
    }

    if (IS_TZ_MEMORY(addr) || IS_TZ_MEMORY(addr + len - 1) ) && addr < 0x2A03F000) {
        return -1;
    }
}
```

<https://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html>

- TEE ranges can be dynamic (and scattered)
 - Hardcoded values may be difficult to handle
- Logical mistakes may happen....

Range checks not so easy...

- Multiple memories:
 - Not everything is DDR
- Layout can be **dynamic**:
 - Example: Video Memory
- Proper check location and API design are fundamental
- **System-level consistency** of view is needed for proper enforcement:
 - Across every SW runtime component
 - Across the whole SoC HW.

“Space” dimension:

Not just the ARM CPU

Remember?

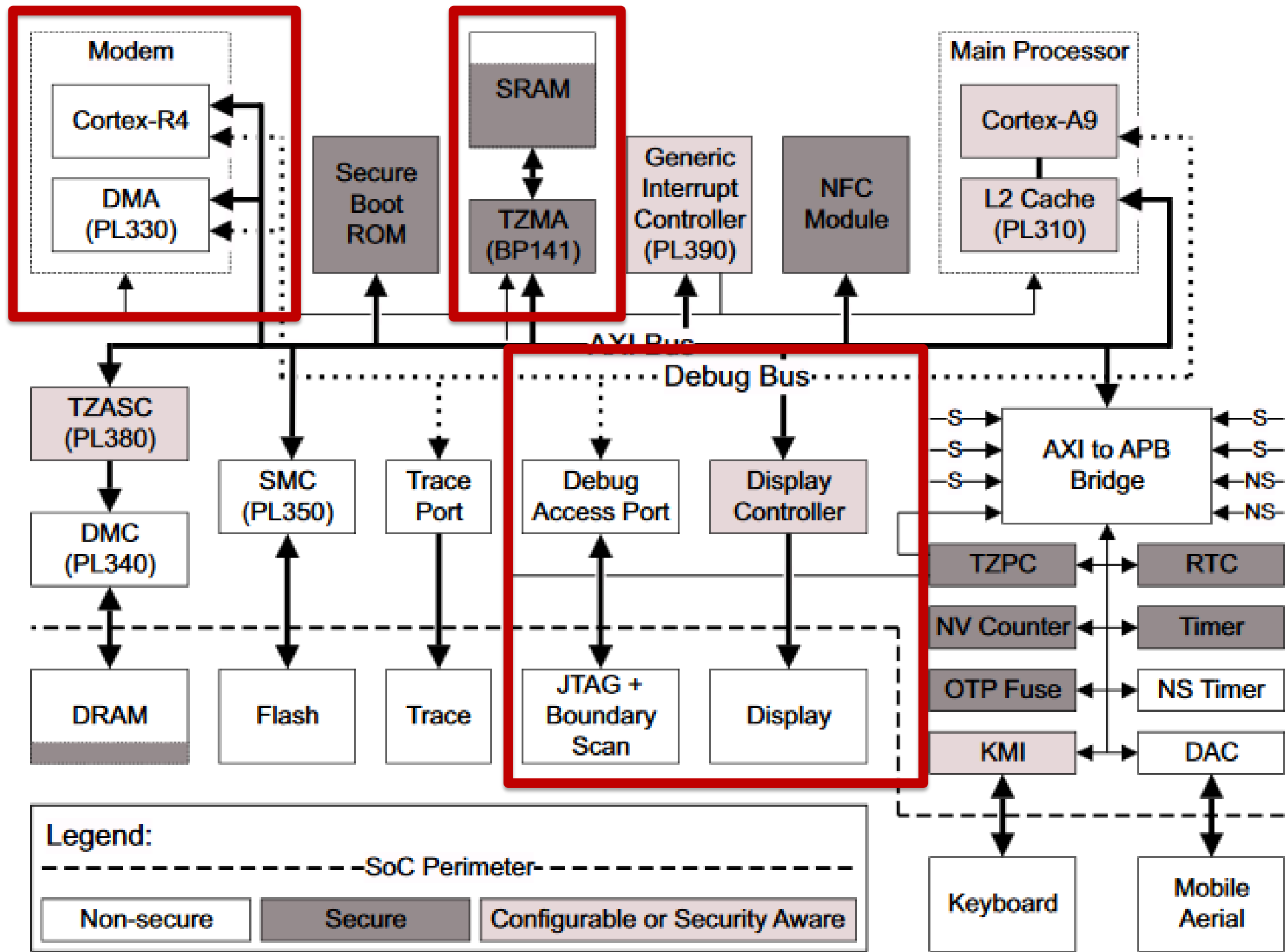


Figure 6-1 : The Gadget2008 SoC design

Potential attack surface

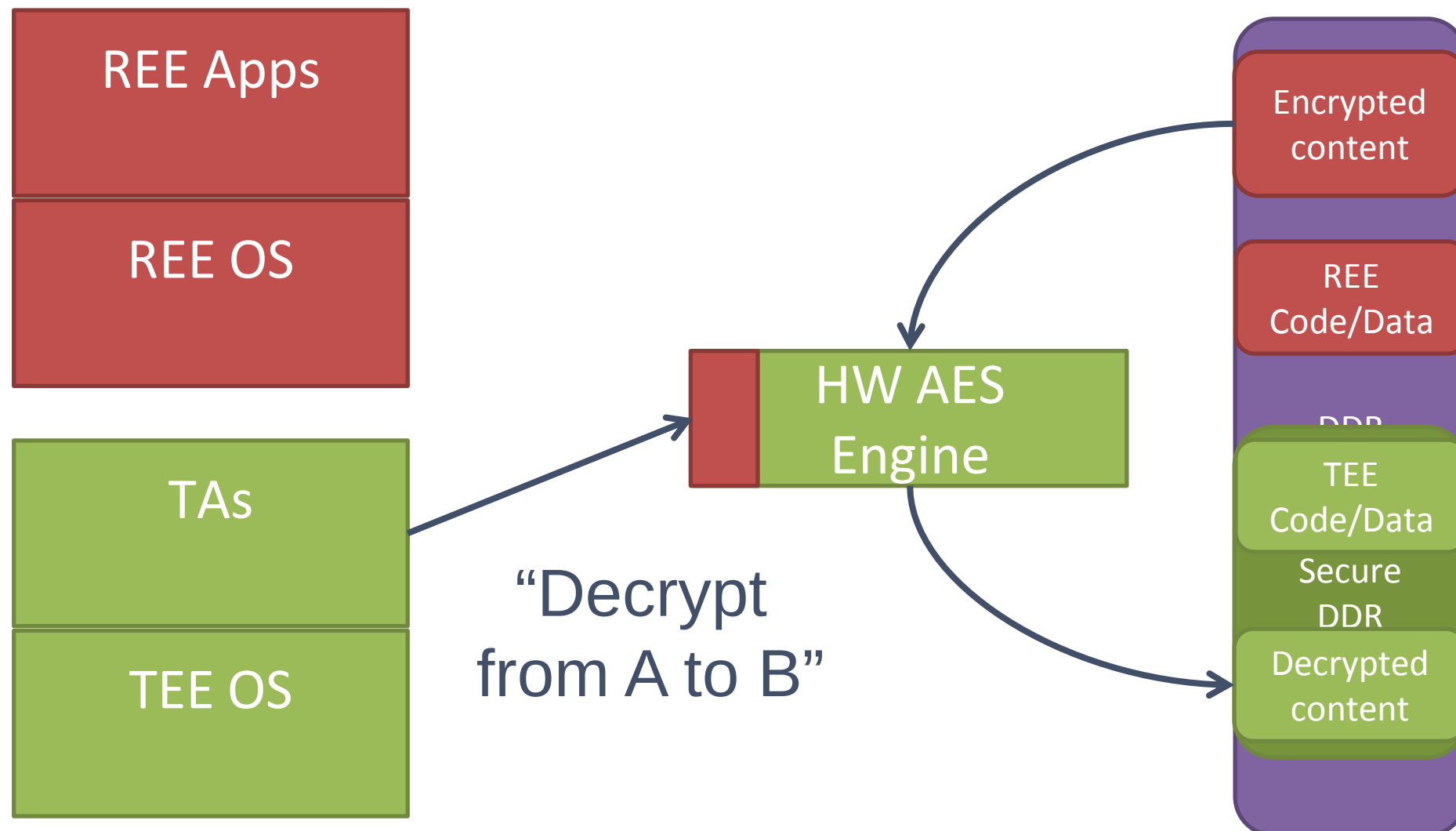
- SoC much more than the ARM CPU
- DMA engines
 - Crypto accelerators
 - PCI/PCIe devices
- Other processing engines
 - Audio/Video CPUs
 - Modem and WiFi controllers
 - Power management MCUs

***Any IP with access to the bus MUST
be considered!***

Buses, masters and slaves

- Most masters are also slaves
 - DMA transactions configured through the bus
 - Auxiliary CPUs expose APIs through the bus
 - ...
- Need to take care of configuration
 - Secure bus masters should not be driven by non-secure processing engines
 - Firmware running on secure bus masters should be authenticated and secured!

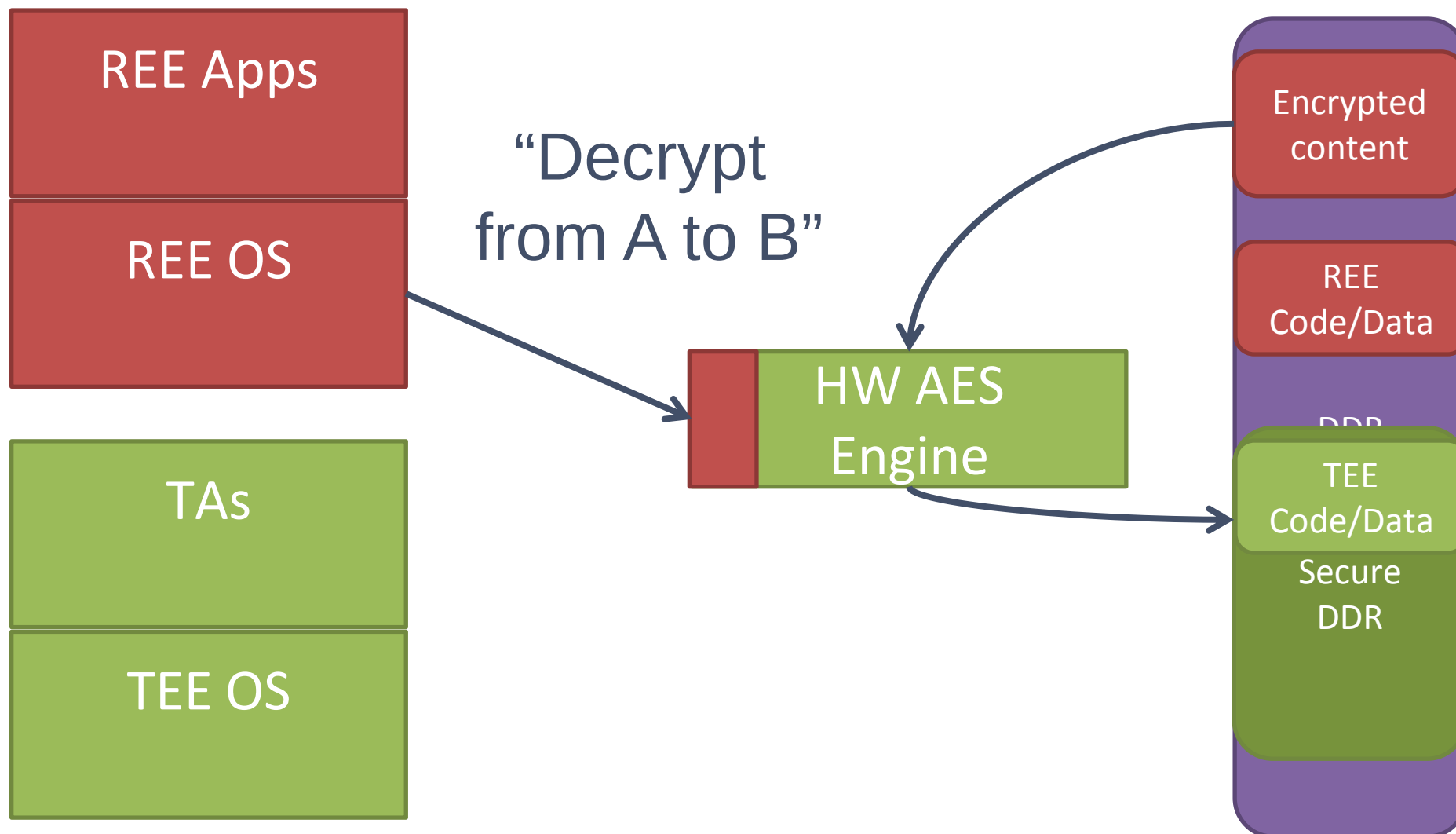
Example: HW crypto engine



Non-secure

Secure

What if... ?



Non-secure

Secure

Securing peripherals

- Some use cases might require isolating peripherals
 - Secure display to show mobile payment data
 - Secure touch sensor for PIN entry
 - Secure fingerprint sensor
- But some peripherals need to be available to both worlds
 - Runtime configuration required

State transitions must be carefully considered

“Time and Space”:

TEE Warm Boot

Warm Boot

- Simply put: *Boot after “Suspend-To-RAM”*
 - Typically requested from REE
- Only some parts of the SoC are powered down:
 - DDR in self-refresh mode
 - Some limited parts always-on for restore
- Restore/reuse saved execution contexts
 - E.g: Entry points

What if...

- Contexts are not fully stored in TEE memory?
- Protection controllers are shutdown as well?
- Contexts are stored in non-DDR memory?
 - E.G. some on-chip SRAM
- Remaining execution cores are non-secure?
 - Do they have access to memory storing contexts?

Conclusion

Conclusion

- TEE security can be complex:
 - Full HW & SW cooperation continuously required
- *TEE initialization is critical*
- *HW can also be an attacker...*
- More accurate ***TEE security model*** needed:
 - Properly frame attacks, discussions and design choices
- ***Holistic*** view required

*TEE is an environment...
...not “just” a feature.*



Thank you!!

Cristofaro Mune (@pulsoid)
pulsoid@icysilence.org

Eloi Sanfelix (@esanfelix)
eloi@riscure.com