

Trusted firmware for A-profile systems

Secure/Realm world interfaces

Trainer: Sumit Garg
Linaro Support and Solutions Engineering



Linaro
Developer Services

Logistics

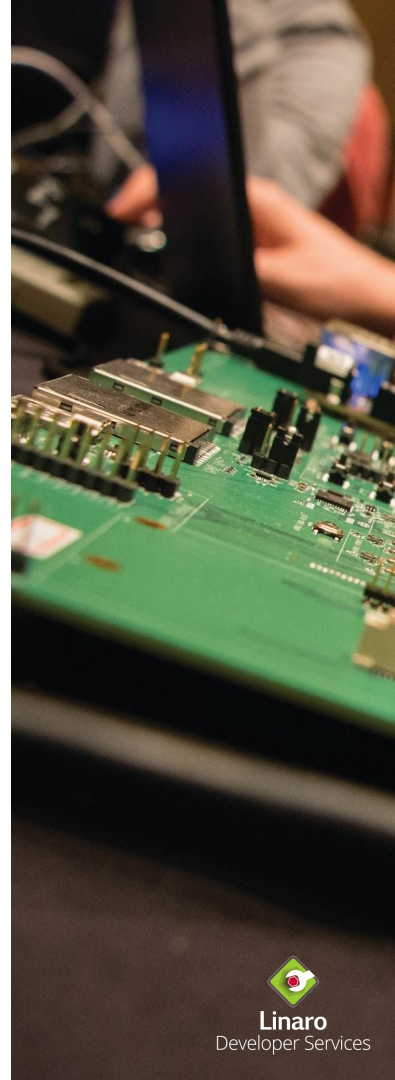
These sessions will be recorded. Turn off your camera if you do not want to appear in the recording.

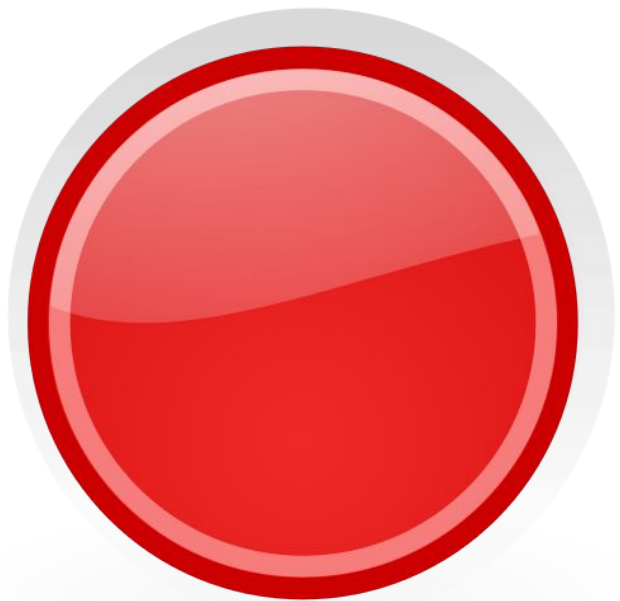
Questions welcome! There is time allocated for Q&A at the end of today's session but you can ask relevant questions verbally or in the chat as we go.

Please keep your microphone muted when not speaking!

Slides and lab resources can be downloaded from:

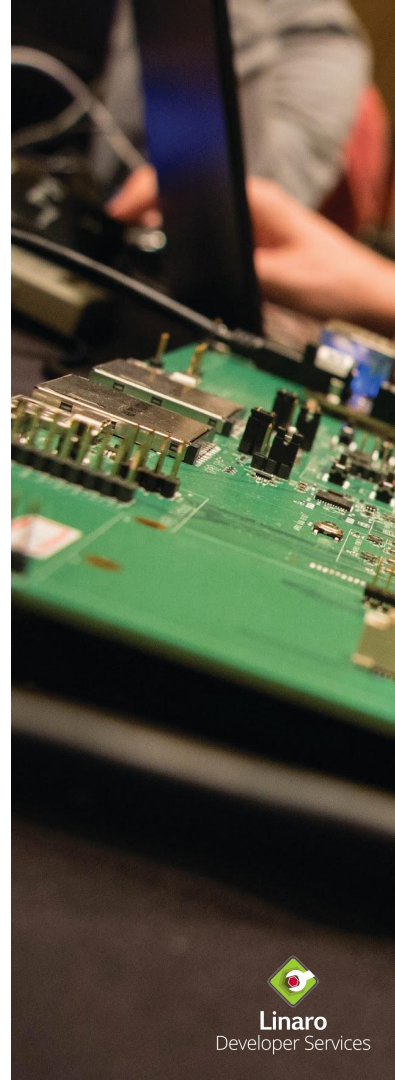
<https://fileserver.linaro.org/s/fE6iBYca8bYqrrk>





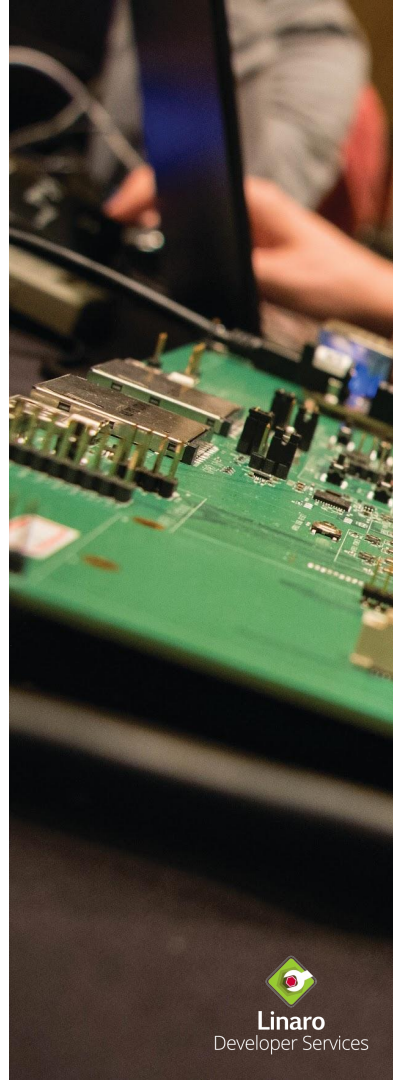
Training modules overview

1. Introduction to TF-A
2. Generic boot
3. Firmware security
- 4. Secure/Realm world interfaces**



Secure/Realm world interfaces

- **EL3 runtime interface (firmware)**
 - Arm architecture services
 - Standard services
 - SiP/OEM specific services
- **EL3 runtime interface (Secure world)**
 - Secure-EL1 payload dispatcher
 - Interrupt handling
 - FF-A standard protocol
 - Secure Partition Manager (SPM)
- **EL3 runtime interface (Realm world)**
 - Realm Management Monitor (RMM)
 - Granule Protection Tables Library



EL3 runtime services

BL31 is the **most privileged** exception level where the **runtime resident** monitor firmware executes. TF-A provides **reference implementation** for EL3 monitor firmware.

Provide **runtime services**, such as:

- Arm architectural services
- Standard services such as **PSCI, SDEI, MM, TRNG** etc.
- SiP/OEM specific services
- Foundation to build:
 - Trusted Execution Environment (**TEE**)
 - **Realms** for Confidential Compute Architecture (**CCA**)

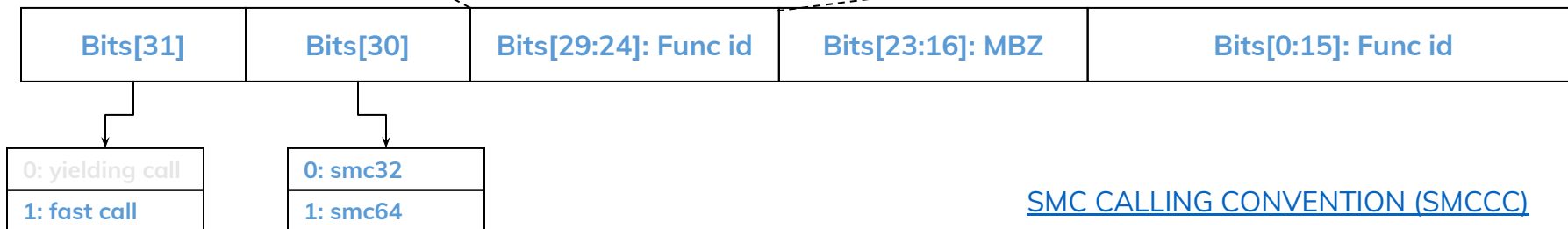
to lower exception levels (EL1/EL2 and S-EL1/S-EL2)

- **EL1 -> Rich OS** or **EL2 -> Hypervisor**
- **S-EL1 -> Trusted OS** or **S-EL2 -> Secure hypervisor**



Arm architectural and standard services: Fast SMC

Entity Number	Bit Mask	Description
0	0x00000000	ARM Architecture Calls
1	0x01000000	CPU Service Calls
2	0x02000000	SIP Service Calls
3	0x03000000	OEM Service Calls
4	0x04000000	Standard Service Calls
5 - 47	0x05000000 – 0x2F000000	Reserved for future use
48 - 49	0x30000000 – 0x31000000	Trusted Application Calls
50 - 63	0x32000000 – 0x3F000000	Trusted OS Calls



[SMC CALLING CONVENTION \(SMCCC\)](#)

Arm architectural services

One of the major user for Arm architectural services is the [firmware interface for mitigating cache speculation vulnerabilities](#).

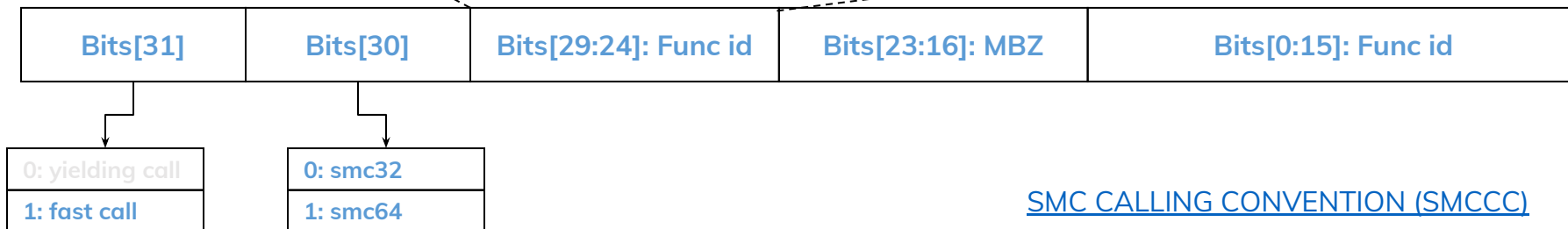
The vulnerabilities mitigated via this interface are:

- **CVE-2017-5715:** Involves invalidation of the branch predictor on Arm Cortex CPUs.
- **CVE-2018-3639:** Involves disabling the bypassing of writes by reads (including speculative reads), either permanently during CPU initialization, or dynamically as required.
- **CVE-2022-23960 aka Spectre-BHB:** Involves flushing all branch predictions via an implementation specific route.



Arm architectural and standard services: Fast SMC

Entity Number	Bit Mask	Description
0	0x00000000	ARM Architecture Calls
1	0x01000000	CPU Service Calls
2	0x02000000	SIP Service Calls
3	0x03000000	OEM Service Calls
4	0x04000000	Standard Service Calls
5 - 47	0x05000000 – 0x2F000000	Reserved for future use
48 - 49	0x30000000 – 0x31000000	Trusted Application Calls
50 - 63	0x32000000 – 0x3F000000	Trusted OS Calls



[SMC CALLING CONVENTION \(SMCCC\)](#)

Standard services

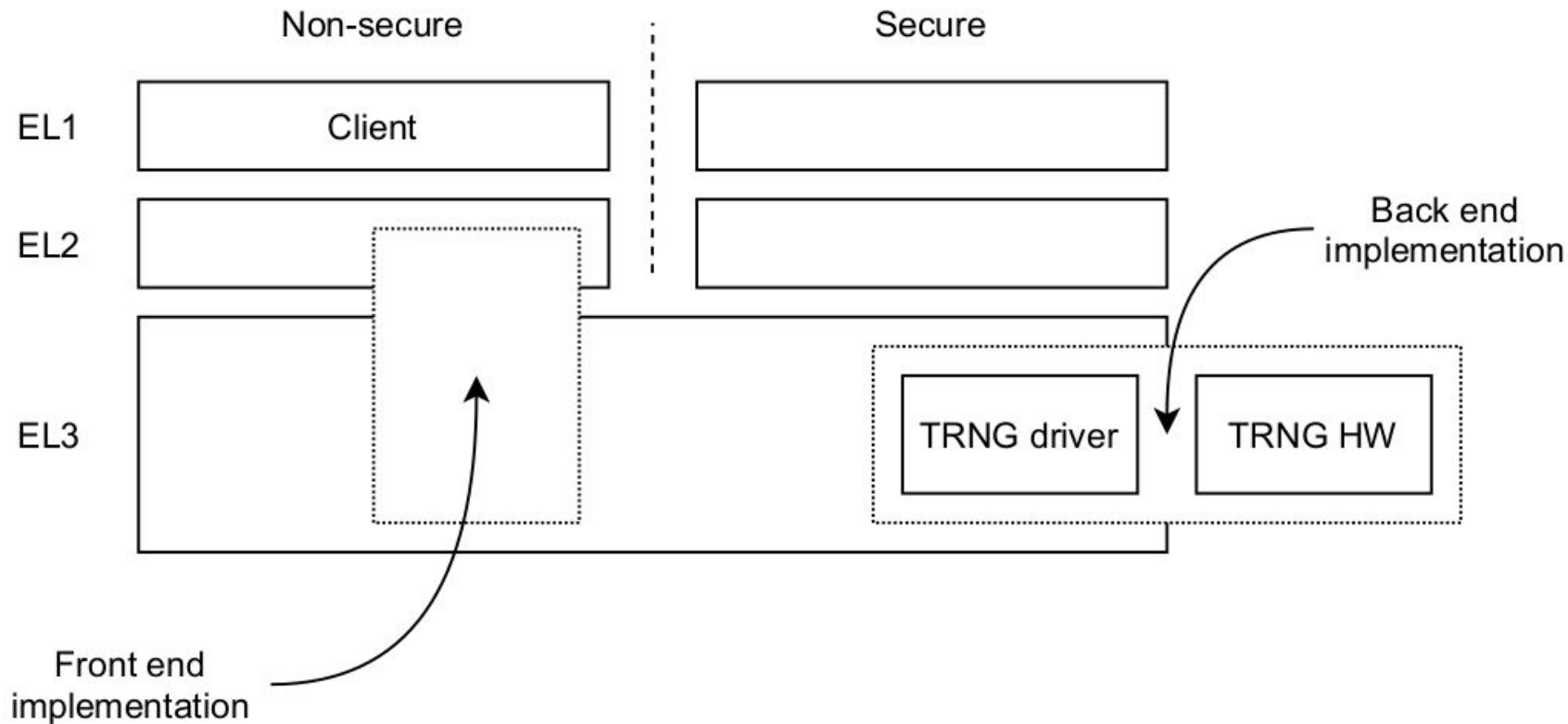
Arm defines a set of **Standard** Secure Service Calls for the **management of the overall system**. Standard calls are intended to provide system management services to operating systems.

Reserved standard services:

- Power Secure Coordination Interface (**PSCI**)
- **TRNG FW** Interface
- **Errata** FW interface
- Management Mode (**MM**) interface
- Software Delegated Exception Interface (**SDEI**)
- PSA Firmware Framework A (**FF-A**) interface
- Arm **CCA** interface



TRNG FW Interface



TRNG FW Interface - II

True Random Number Generator (TRNG) firmware interface provides a **conditioned entropy source** to an Operating System (OS). The conditioned entropy is commonly used to seed deterministic random number generators (**DRNG**) or to **generate keys**, among other use-cases.

TRNG FW ABI:

- **TRNG_VERSION**: returns the implemented TRNG ABI **version**.
- **TRNG_FEATURES**: determine if functions defined in the TRNG ABI are **present** in the ABI implementation.
- **TRNG_GET_UUID**: returns the **UUID** of the TRNG Back end.
- **TRNG_RND**: returns **N bits of conditioned entropy** if successful, otherwise returns immediately with an error code.

Errata FW interface

Errata describe hardware features which **deviate from the design intent**. An erratum can have an **associated workaround**, implementable in software, to mitigate the erratum.

- Some workarounds are implementable at **different ELs**.
- Some workarounds may require actions to be taken at **multiple ELs** to fully mitigate the erratum.

A **CPU IP erratum** is identified by the **CPU_erratum_ID** identifier. The CPU_erratum_ID is a core IP vendor specified **32 bit value** that unambiguously identifies the erratum on a particular core. A **disclosed** erratum must specify the CPU_erratum_ID and the core that it relates to.



Errata FW interface - II

Errata FW interface provides OS the **capability to discover the CPU errata** that it must deploy mitigations for.

- Discover the errata known to higher ELs and that have been **fixed in hardware or mitigated at a higher EL**.
- Discover the errata which **require mitigation by the OS**.

Errata FW ABI:

- **EM_VERSION**: returns the implemented version of the Errata ABI
- **EM_FEATURES**: discover the Errata ABI functions that are implemented in the firmware.
- **EM_CPU_ERRATUM_FEATURES**: obtains the features of a given CPU erratum. These features describe whether software at the calling EL or lower can be affected by an erratum.

Management Mode (MM) interface

Management Mode (MM) provides an **environment** for implementing **OS agnostic services** (MM services) like:

- **RAS** error handling
- **Secure variable storage**
- **Firmware updates** in system firmware

MM interface provides the backend implementation for **standardized UEFI interface** towards OS. The **UEFI PI spec** describes the functionality and how that should be accessed.

ABI:

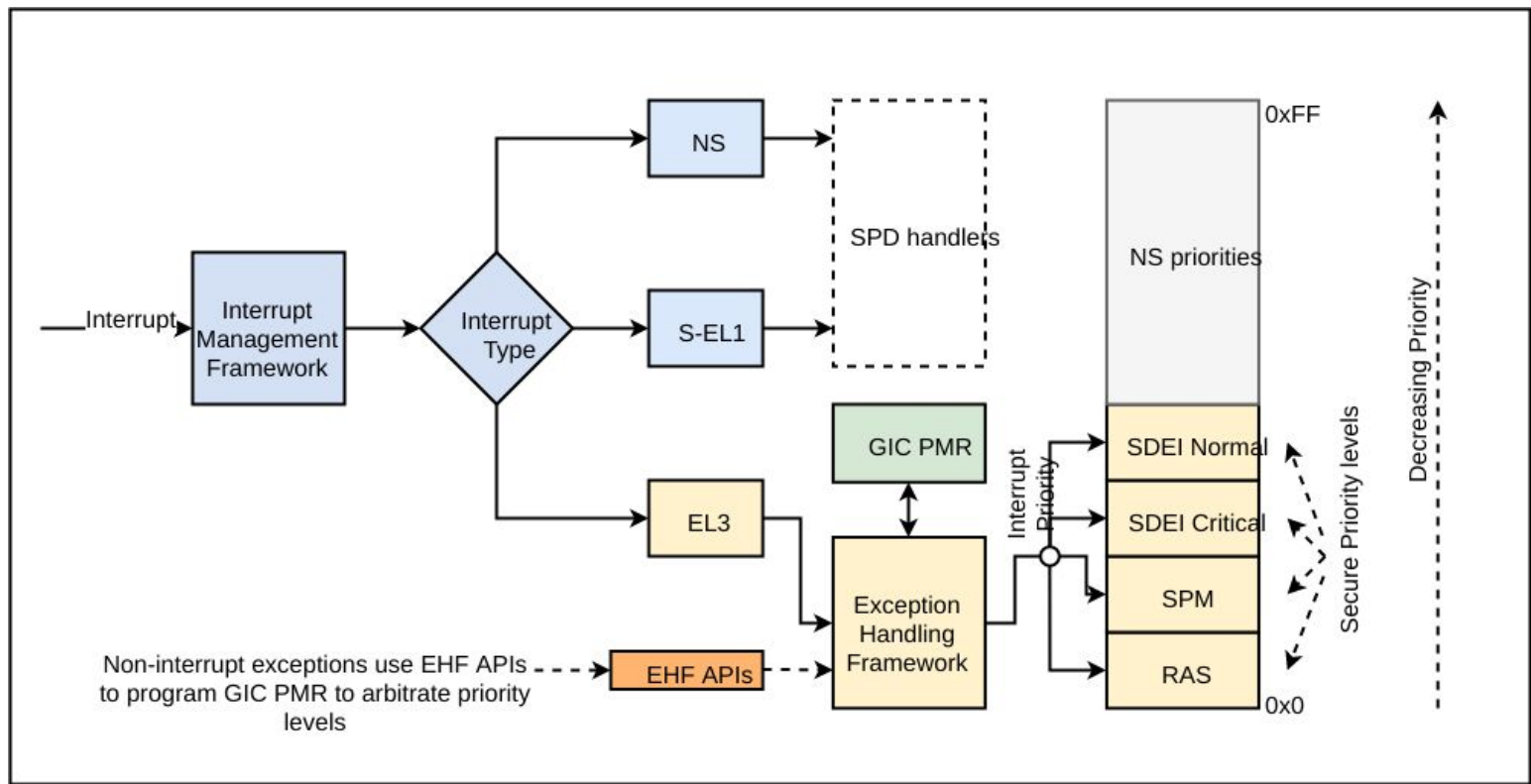
- **MM_VERSION**: returns the version of the MM implementation.
- **MM_COMMUNICATE**: invokes an MM service.

Exception handling framework (SDEI)

- Exception handling framework changes the semantics the interrupts and exceptions that are targeted at EL3
 - EHF is enabled with option 'EL3_EXCEPTION_HANDLING=1'
- EHF provides a *Firmware-first Handling* approach
 - Response RAS events at the earliest time
 - Delegate message to the normal world, even the normal world has masked exceptions
- EHF partitions priority levels
 - Interrupt or an exception is assigned to a priority level and associated to a dispatcher
 - Don't mix the EHF dispatcher with SPD dispatcher (two different things!)



Role of exception handling framework

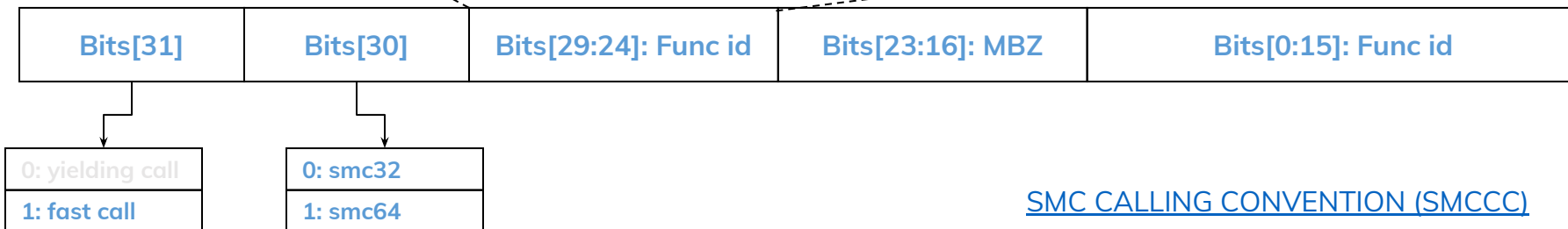


<https://trustedfirmware-a.readthedocs.io/en/latest/components/exception-handling.html>

For Qualcomm - July 2023

SiP/OEM specific services: Fast SMC

Entity Number	Bit Mask	Description
0	0x00000000	ARM Architecture Calls
1	0x01000000	CPU Service Calls
2	0x02000000	SIP Service Calls
3	0x03000000	OEM Service Calls
4	0x04000000	Standard Service Calls
5 - 47	0x05000000 – 0x2F000000	Reserved for future use
48 - 49	0x30000000 – 0x31000000	Trusted Application Calls
50 - 63	0x32000000 – 0x3F000000	Trusted OS Calls



[SMC CALLING CONVENTION \(SMCCC\)](#)

SiP/OEM specific services

SiP/OEM services are **non-standard**, platform-specific services offered by the **silicon implementer** or **platform provider**.

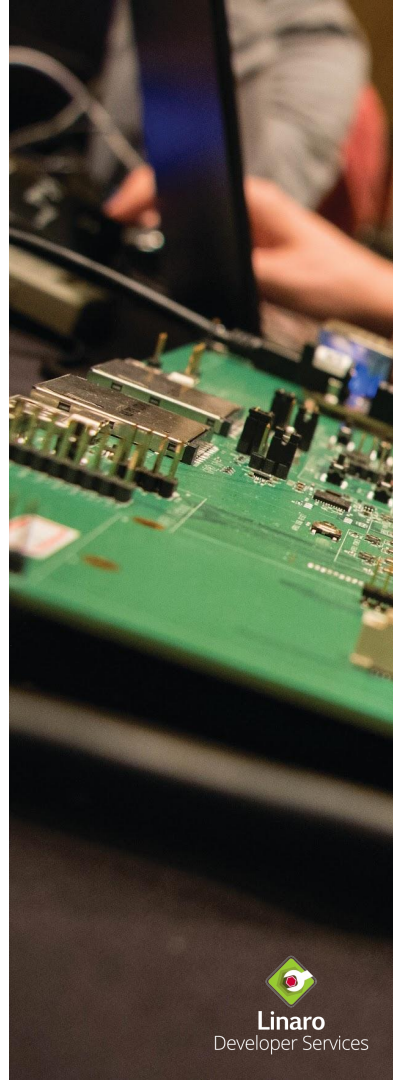
The **Arm SiP** implementation offers the following services:

- **Performance Measurement Framework (PMF)**: allows callers to retrieve timestamps captured at various paths in TF-A execution.
- **Execution State Switching service**: provides a mechanism for a non-secure lower Exception Level to request to switch its execution state, either from **AArch64 to AArch32**, or from **AArch32 to AArch64**, for the calling CPU.
- **DebugFS** interface: primarily aimed at exposing **firmware debug data** to higher SW layers such as a non-secure component.

Other silicon vendors like **Amlogic**, **Xilinx (AMD)** etc. do provide platform specific SiP services.

Secure/Realm world interfaces

- EL3 runtime interface (firmware)
 - Arm architecture services
 - Standard services
 - SiP/OEM specific services
- **EL3 runtime interface (Secure world)**
 - **Secure-EL1 payload dispatcher**
 - Interrupt handling
 - FF-A standard protocol
 - Secure Partition Manager (SPM)
- EL3 runtime interface (Realm world)
 - Realm Management Monitor (RMM)
 - Granule Protection Tables Library



EL3 runtime services

BL31 is the **most privileged** exception level where the **runtime resident** monitor firmware executes. TF-A provides **reference implementation** for EL3 monitor firmware.

Provide **runtime services**, such as:

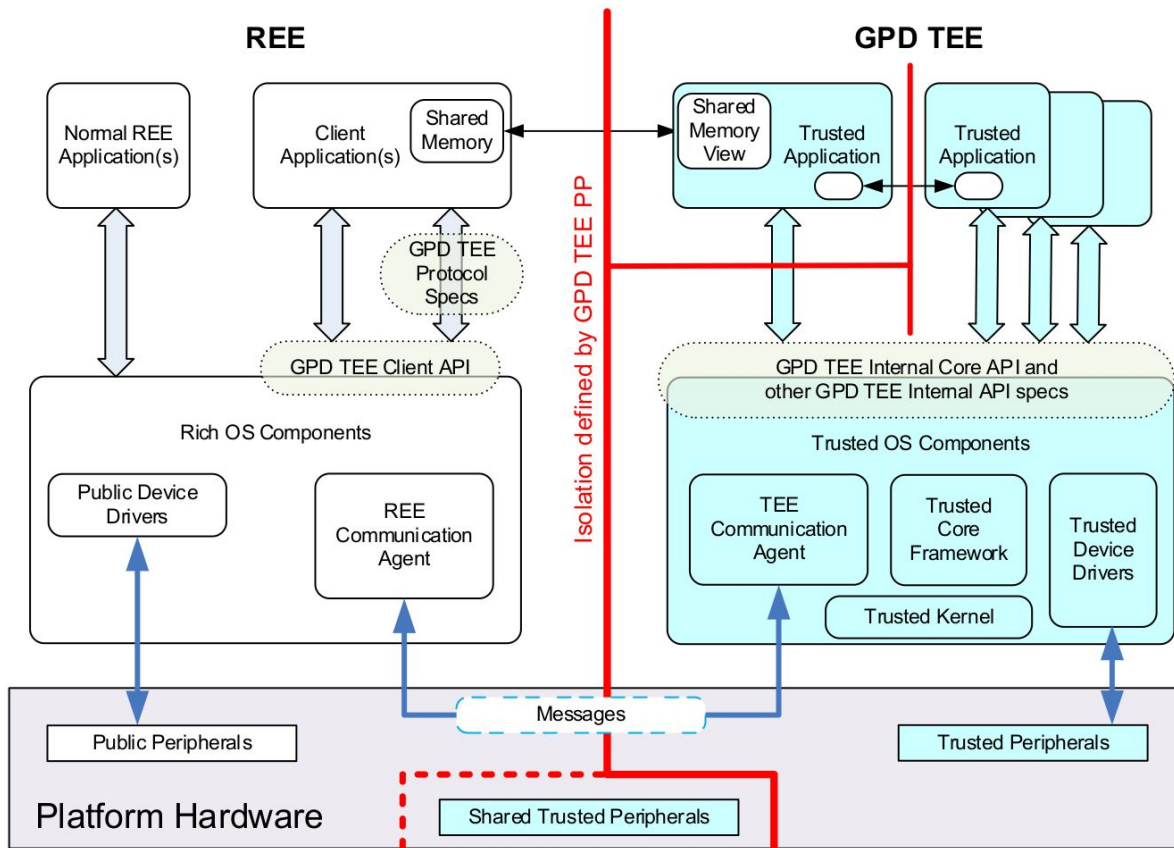
- Arm architectural services
- Standard services such as **PSCI, SDEI, MM, TRNG** etc.
- SiP/OEM specific services
- Foundation to build:
 - Trusted Execution Environment (**TEE**)
 - **Realms** for Confidential Compute Architecture (**CCA**)

to lower exception levels (EL1/EL2 and S-EL1/S-EL2)

- **EL1 -> Rich OS** or **EL2 -> Hypervisor**
- **S-EL1 -> Trusted OS** or **S-EL2 -> Secure hypervisor**



Generic TEE architecture

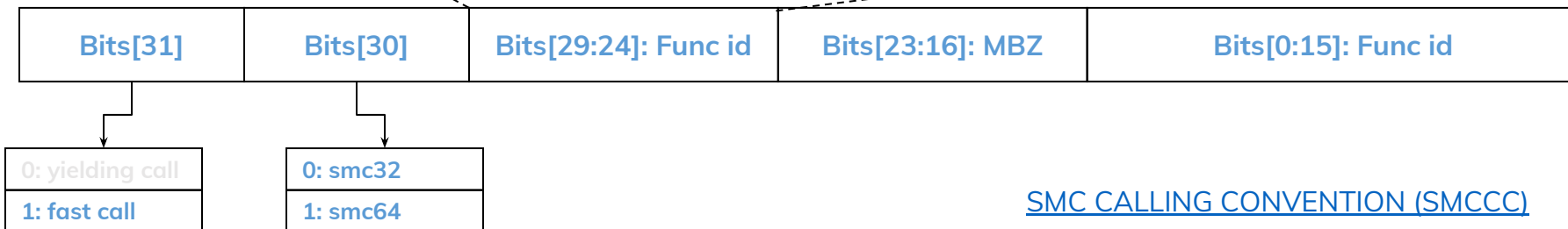


GPD TEE:
GlobalPlatform
Defined TEE

Source:
GPD_TEE_SystemArch_v1.2
PublicRelease

TEE messages via fast SMC

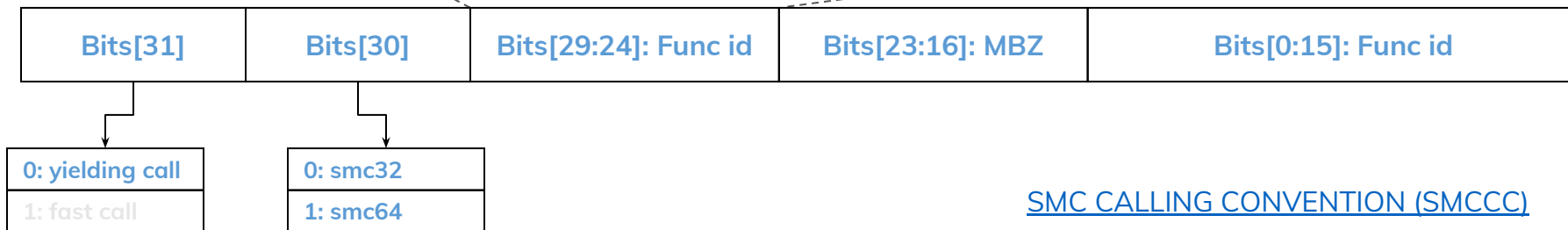
Entity Number	Bit Mask	Description
0	0x00000000	ARM Architecture Calls
1	0x01000000	CPU Service Calls
2	0x02000000	SIP Service Calls
3	0x03000000	OEM Service Calls
4	0x04000000	Standard Service Calls
5 - 47	0x05000000 – 0x2F000000	Reserved for future use
48 - 49	0x30000000 – 0x31000000	Trusted Application Calls
50 - 63	0x32000000 – 0x3F000000	Trusted OS Calls



[SMC CALLING CONVENTION \(SMCCC\)](#)

TEE messages via yielding SMC

Entity Number	Bit Mask	Description
0-1	0x00000000 – 0x0100FFFF	Reserved because this region is already in use by ARMv7 devices on the field. Strictly speaking this is reserved only when Bit[30] is set for smc32.
2-31	0x02000000 – 0x1FFFFFFF	Trusted OS Calls
32-63	0x20000000 – 0x3F000000	Reserved for future expansion of Trusted OS Yielding Calls



[SMC CALLING CONVENTION \(SMCCC\)](#)

Secure-EL1 payload dispatcher

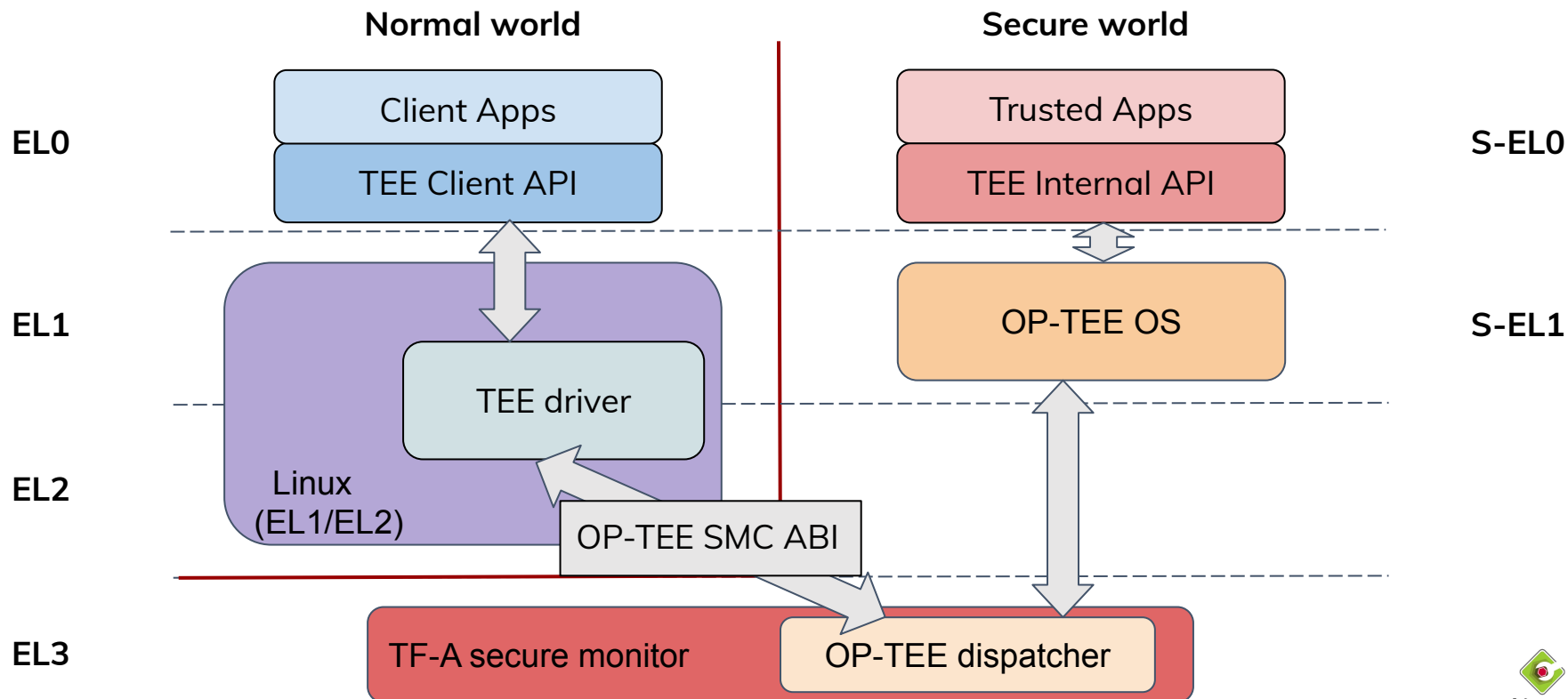
Secure-EL1 payload dispatcher entity within EL3 firmware responsible for the **initialisation** of the Trusted OS and all **communications** with it. The **Trusted OS** is the BL32 stage of the boot flow in TF-A.

TF-A provides a **Test Secure-EL1 Payload (TSP)** and a Test Secure-EL1 Payload Dispatcher (**TSPD**) service as an **example** of how a Trusted OS is supported on a production system using the Runtime Services Framework.

Upstream supported Trusted OSes:

- **OP-TEE OS** and dispatcher as **opteed**, a Trusted Firmware community project.
- **Google Trusty** and dispatcher as **trusty**.
- **Nvidia Trusted Little Kernel (TLK)** and dispatcher as **tlkd**.
- **ProvenCore micro-kernel** and dispatcher as **pncd**.

Example: OP-TEE dispatcher (opteed)



Example: OP-TEE runtime service descriptor

```
typedef int32_t (*rt_svc_init_t)(void);

typedef uintptr_t (*rt_svc_handle_t)(uint32_t smc_fid,
                                     u_register_t x1, u_register_t x2,
                                     u_register_t x3, u_register_t x4,
                                     void *cookie, void *handle,
                                     u_register_t flags);

typedef struct rt_svc_desc {
    uint8_t start_oen;
    uint8_t end_oen;
    uint8_t call_type;
    const char *name;
    rt_svc_init_t init;
    rt_svc_handle_t handle;
} rt_svc_desc_t;

#define DECLARE_RT_SVC(_name, _start, _end, _type, _setup, _smch) \
    ...
```



```
/* OPTeED runtime service descriptor for fast SMC calls */
DECLARE_RT_SVC(
    opteed_fast,

    OEN_TOS_START,
    OEN_TOS_END,
    SMC_TYPE_FAST,
    opteed_setup,
    opteed_smc_handler
);

/* OPTeED runtime service descriptor for yielding SMC calls
*/
DECLARE_RT_SVC(
    opteed_std,

    OEN_TOS_START,
    OEN_TOS_END,
    SMC_TYPE_YIELD,
    NULL,
    opteed_smc_handler
);
```



Example: OP-TEE SMC handler - request

```
uint64_t opteed_smc_handler(uint32_t smc_fid, ...) {
    optee_context_t *optee_ctx = &opteed_sp_context[plat_my_core_

    if (is_caller_non_secure(flags)) {
        cm_el1_sysregs_context_save(NON_SECURE);
        cm_set_elr_el3(SECURE, GET_SMC_TYPE(smc_fid) == SMC_TYPE_FAST ?
                        &optee_vectors->fast_smc_entry :
                        &optee_vectors->std_smc_entry;
        cm_el1_sysregs_context_restore(SECURE);
        cm_set_next_eret_context(SECURE);
        write_ctx_reg(get_gpregs_ctx(&optee_ctx->cpu_ctx),
                      CTX_GPRREG_X4, read_ctx_reg(get_gpregs_ctx(handle),
                                                    CTX_GPRREG_X4));
        /* also copy X5-X7 from get_gpregs_ctx(handle) to optee_ctx */
        SMC_RET4(&optee_ctx->cpu_ctx, smc_fid, x1, x2, x3);
    }
}
```

optee_vectors is
initialized by OP-TEE
(S-EL1 payload)

...

Example: OP-TEE OS handling yielding call

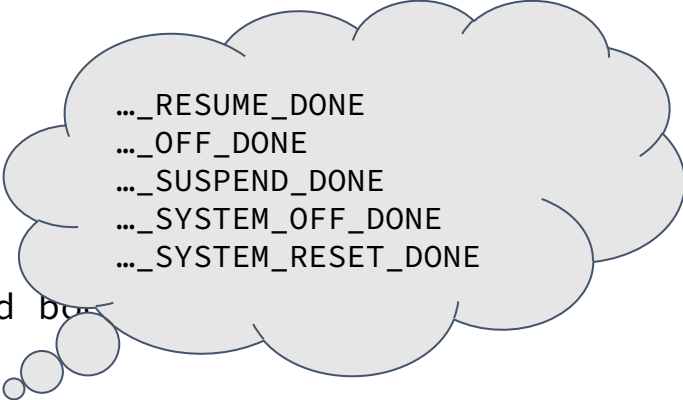
```
uint32_t __tee_entry_std(struct optee_msg_arg *arg, uint32_t num_params)
{
    uint32_t rv = OPTEE_SMC_RETURN_OK;

    /* Enable foreign interrupts for STD calls */
    thread_set_foreign_intr(true);
    switch (arg->cmd) {
    case OPTEE_MSG_CMD_OPEN_SESSION:
        entry_open_session(arg, num_params);
        break;
    case OPTEE_MSG_CMD_CLOSE_SESSION:
        entry_close_session(arg, num_params);
        break;
```

...

Example: OP-TEE SMC handler return

```
...
switch (smc_fid) {
case TEESMC_OPTEEED_RETURN_ENTRY_DONE:
    optee_vectors = (optee_vectors_t *) x1;
    if (optee_vectors)
        /* finalize initialization after a cold boot */
    opteed_synchronous_sp_exit(optee_ctx, x1);
case TEESMC_OPTEEED_RETURN_ON_DONE: /* other cases share this code path */
    opteed_synchronous_sp_exit(optee_ctx, x1);
case TEESMC_OPTEEED_RETURN_CALL_DONE:
    cm_el1_sysregs_context_save(SECURE);
    ns_cpu_context = cm_get_context(NON_SECURE);
    cm_el1_sysregs_context_restore(NON_SECURE);
    cm_set_next_eret_context(NON_SECURE);
    SMC_RET4(ns_cpu_context, x1, x2, x3, x4);
case TEESMC_OPTEEED_RETURN_FIQ_DONE:
```



- ..._RESUME_DONE
- ..._OFF_DONE
- ..._SUSPEND_DONE
- ..._SYSTEM_OFF_DONE
- ..._SYSTEM_RESET_DONE

Example: Declaring PM services

```
typedef struct spd_pm_ops {  
    void (*svc_on)(u_register_t target_cpu);  
    int32_t (*svc_off)(u_register_t __unused);  
    void (*svc_suspend)(u_register_t max_off_pwrlvl);  
    void (*svc_on_finish)(u_register_t __unused);  
    void (*svc_suspend_finish)(u_register_t max_off_pwrlvl);  
    int32_t (*svc_migrate)(u_register_t from_cpu, u_register_t to_cpu);  
    int32_t (*svc_migrate_info)(u_register_t *resident_cpu);  
    void (*svc_system_off)(void);  
    void (*svc_system_reset)(void);  
} spd_pm_ops_t;
```



Example: OP-TEE CPU off handler

```
static int32_t opteed_cpu_off_handler(uint64_t unused)
{
    int32_t rc = 0;
    optee_context_t *optee_ctx = &opteed_sp_context[plat_my_core_pos()];

    cm_set_elr_el3(SECURE, (uint64_t) &optee_vectors->cpu_off_entry);
    rc = opteed_synchronous_sp_entry(optee_ctx);
    if (rc != 0)
        panic();

    set_optee_pstate(optee_ctx->state, OPTEE_PSTATE_OFF);
    return 0;
}
```

All the PM handlers
share similar structure
and wrap around
opteed_synchronous_
sp_entry.



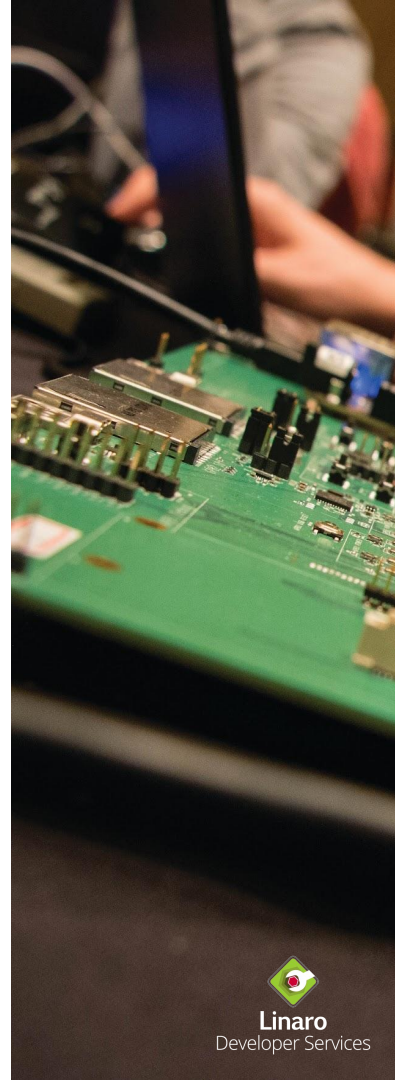
Short break

Based on feedback from previous courses...

- ... we added this to the middle...

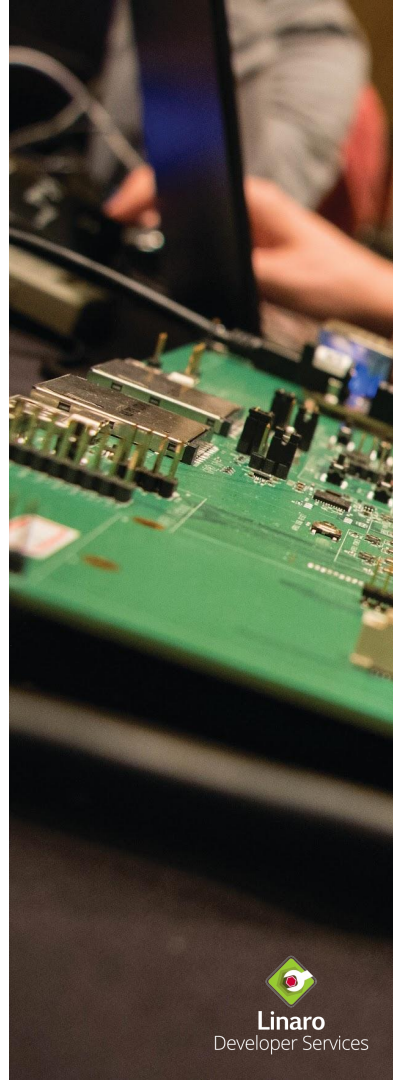
- ... because it is important ...

- ... and after two hours it is hard to recollect bits



Secure/Realm world interfaces

- EL3 runtime interface (firmware)
 - Arm architecture services
 - Standard services
 - SiP/OEM specific services
- **EL3 runtime interface (Secure world)**
 - Secure-EL1 payload dispatcher
 - **Interrupt handling**
 - FF-A standard protocol
 - Secure Partition Manager (SPM)
- EL3 runtime interface (Realm world)
 - Realm Management Monitor (RMM)
 - Granule Protection Tables Library

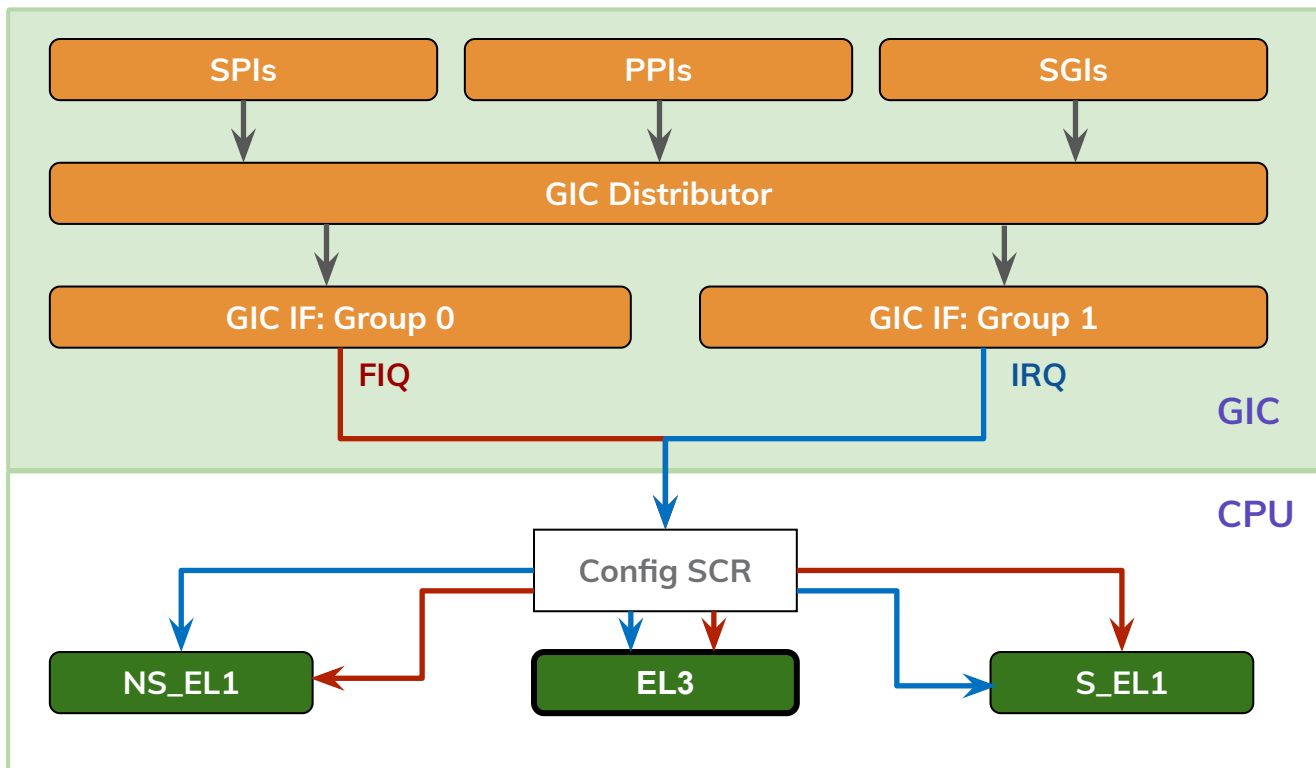


Interrupt handling

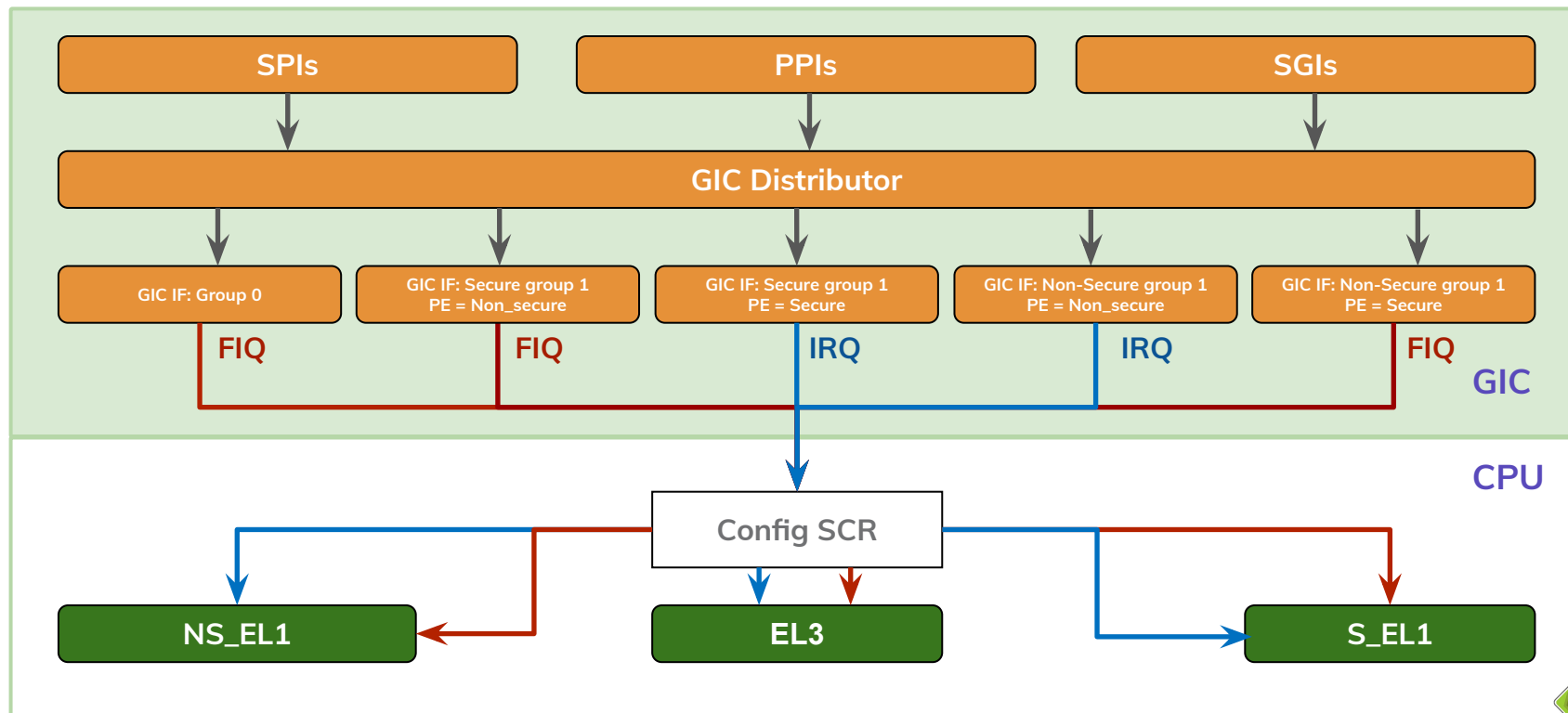
- Three different types of interrupt
 - Interrupt for **EL3** (Not supported by GICv2)
 - Interrupt for **non-secure world**
 - Interrupt for **secure world**
- **Interrupt handling by TF-A is flexible**
 - Supports different threading models within the trusted OS
 - The way the worlds interact is largely dictated by the threading model provided by the trusted OS
- **S-EL1 dispatcher** manages configuration of the interrupt model and an resulting world switches



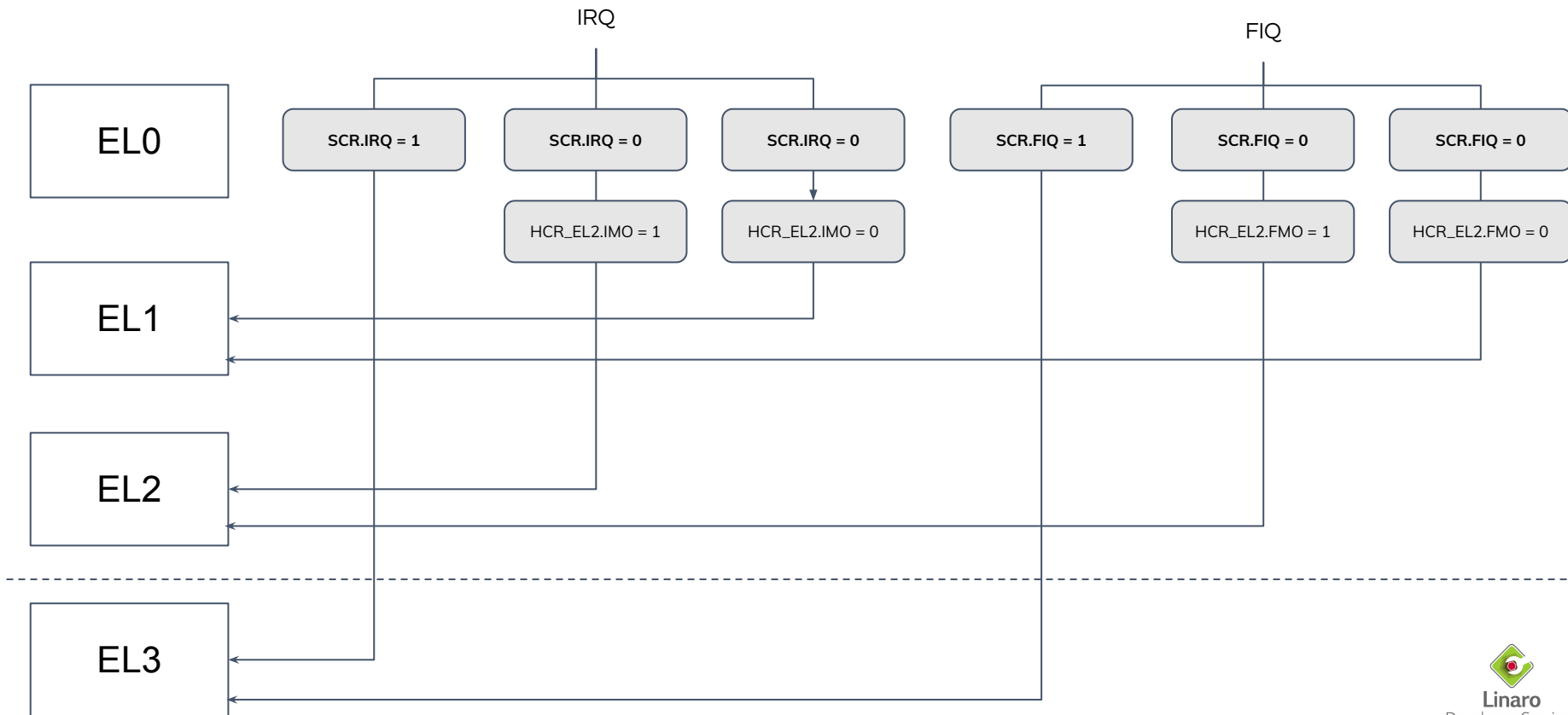
Interrupt hardware routing (GICv2 + ARMv8)



Interrupt hardware routing (GICv3 + ARMv8)

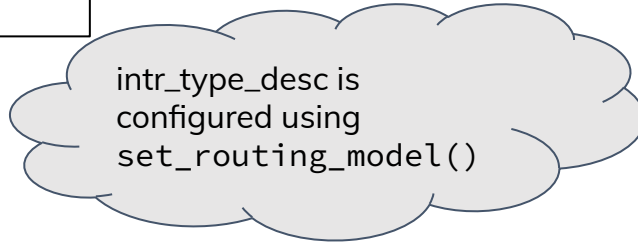


Interrupt taken to exception levels



Interrupt software routing model

```
/*  
 * Register an interrupt handler for S-EL1 interrupts  
 * when generated during code executing in the  
 * non-secure state.  
 */  
flags = 0;  
set_interrupt_rm_flag(flags, NON_SECURE);  
rc = register_interrupt_type_handler(INTR_TYPE_S_EL1,  
                                     opteed_sel1_interrupt_handler,  
                                     flags);
```



FIQ

NS_EL1

EL3

opteed_sel1_interrupt_handler()

```
intr_type_desc[INTR_TYPE_S_EL1].scr_el3[SECURE]   = 0x0  
intr_type_desc[INTR_TYPE_S_EL1].scr_el3[NOSECURE] = SCR_FIQ_BIT
```

Debugging unexpected secure interrupt w/o SPD

```
void bl31_plat_runtime_setup(void)
{
#ifdef SPD_none
    uint32_t flags;
    int32_t rc;

    flags = 0;
    set_interrupt_rm_flag(flags, NON_SECURE);
    rc = register_interrupt_type_handler(INTR_TYPE_S_EL1,
        Hikey_debug_fiq_handler, flags);
    if (rc != 0)
        panic();
#endif
}
```

```
#ifdef SPD_none
static uint64_t hikey_debug_fiq_handler(uint32_t id,
                                         uint32_t flags,
                                         void *handle,
                                         void *cookie)
{
    int intr, intr_raw;

    /* Acknowledge interrupt */
    intr_raw = plat_ic_acknowledge_interrupt();
    intr = plat_ic_get_interrupt_id(intr_raw);
    ERROR("Invalid interrupt: intr=%d\n", intr);
    console_flush();
    panic();

    return 0;
}
#endif
```

Most common interrupt routing model

- Use GIC groups to distinguish interrupts for two worlds
 - Configuration for GICv2
 - Group 0 for secure interrupts with FIQ signal
 - Group 1 for non-secure interrupts with IRQ signal
 - Secure interrupts have higher priority than non-secure interrupts
 - Configuration for GICv3
 - Group 0 for EL3 interrupts
 - Secure group 1 for secure world, non-secure group 1 for non-secure world
- Secure interrupt will preempt non-secure OS
 - Automatically trapped into EL3's vector
 - Secure payload dispatcher will prepare context switch to secure world
 - Once secure interrupt is handled, resume back to non-secure OS
- Non-secure interrupt cannot preempt secure OS
 - Will trap into S-EL1's vector but not EL3's vector
 - Secure OS will decide when return back to non-secure world (possible long latency!)



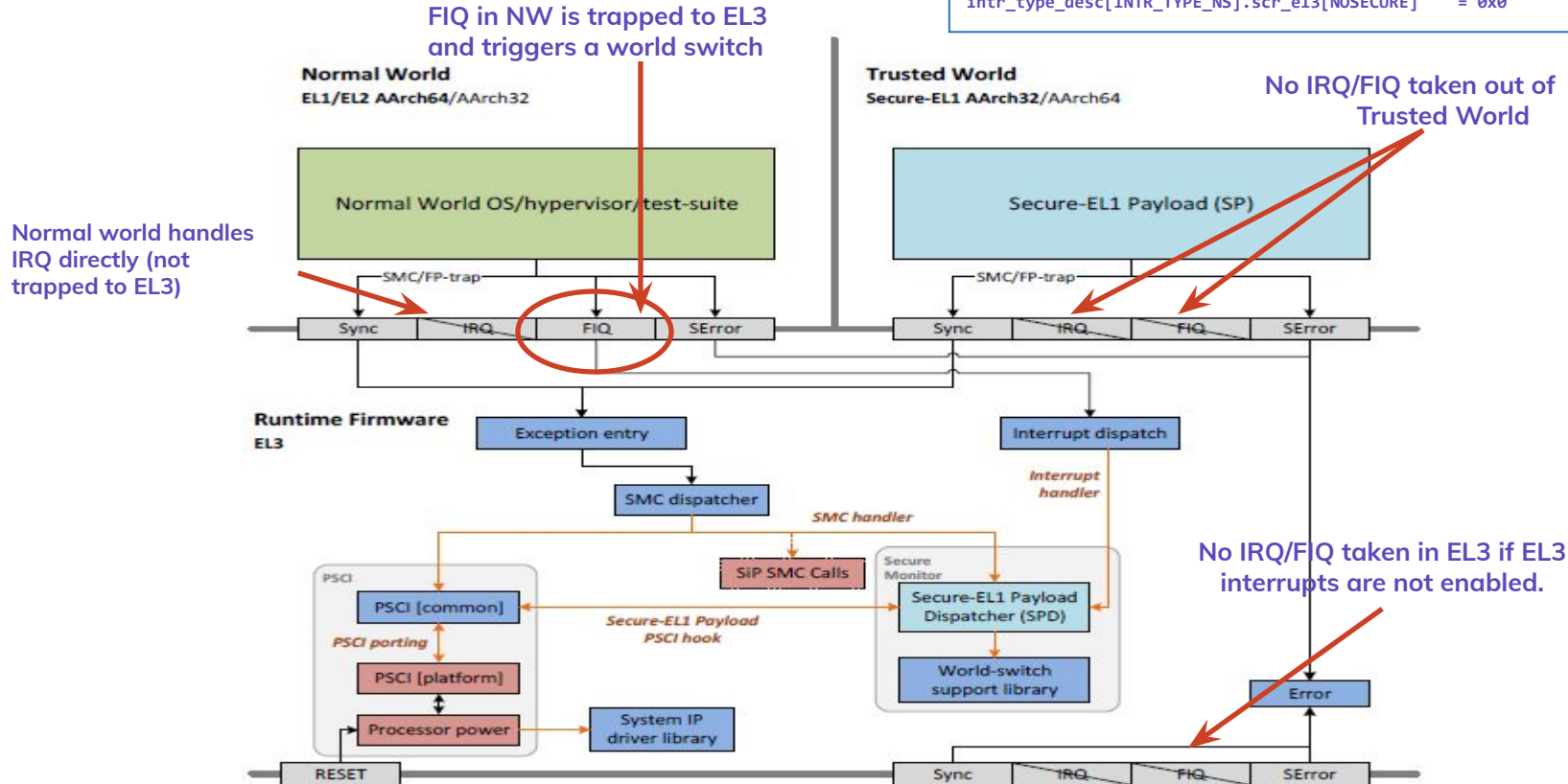
Example of routing

```

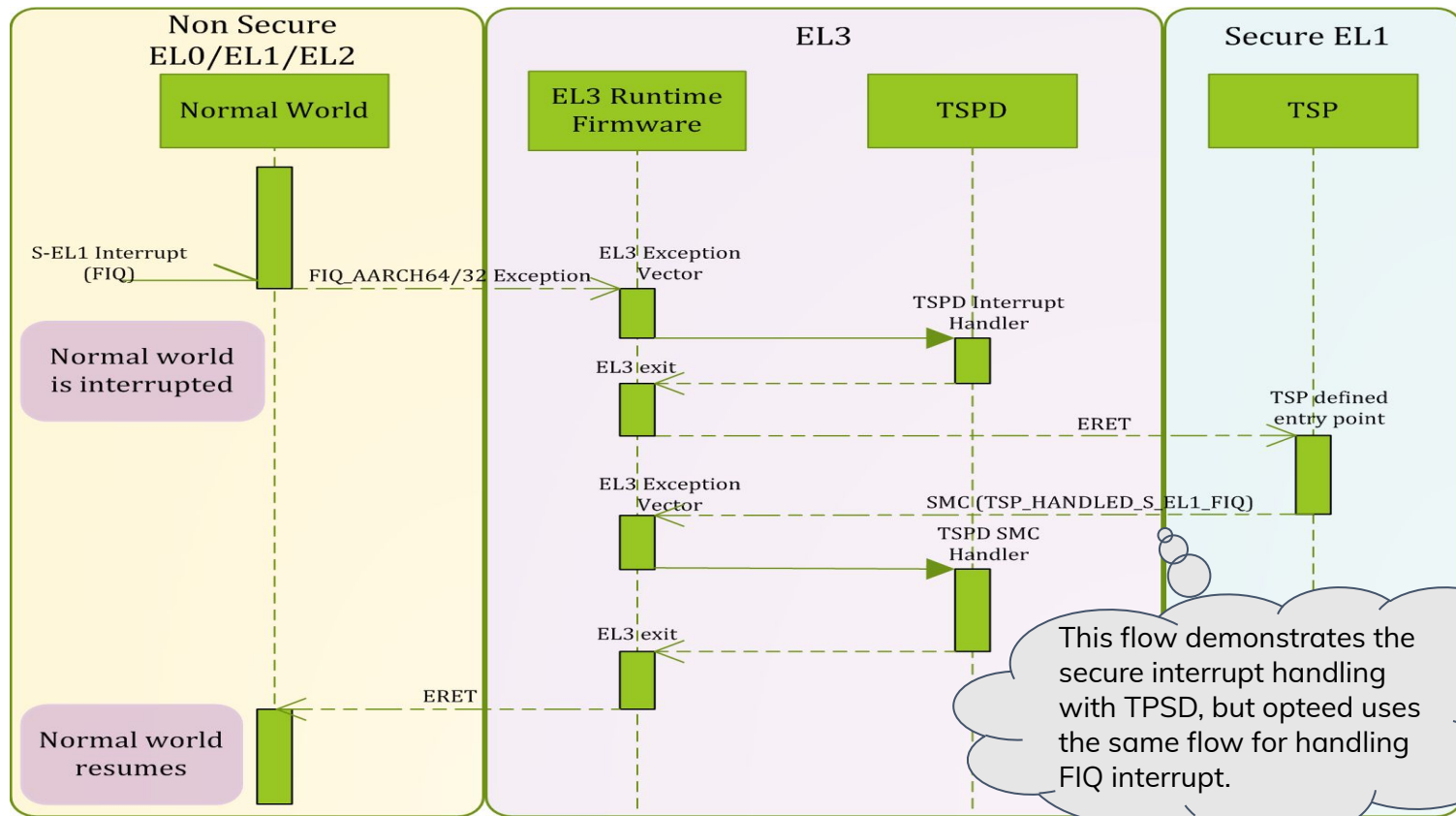
intr_type_desc[INTR_TYPE_EL3].scr_el3[SECURE]    = 0x0
intr_type_desc[INTR_TYPE_EL3].scr_el3[NOSECURE]  = 0x0

intr_type_desc[INTR_TYPE_S_EL1].scr_el3[SECURE]  = 0x0
intr_type_desc[INTR_TYPE_S_EL1].scr_el3[NOSECURE] = SCR_FIQ_BIT
intr_type_desc[INTR_TYPE_S_EL1].handler          = opteed_sel1_interrupt_handler

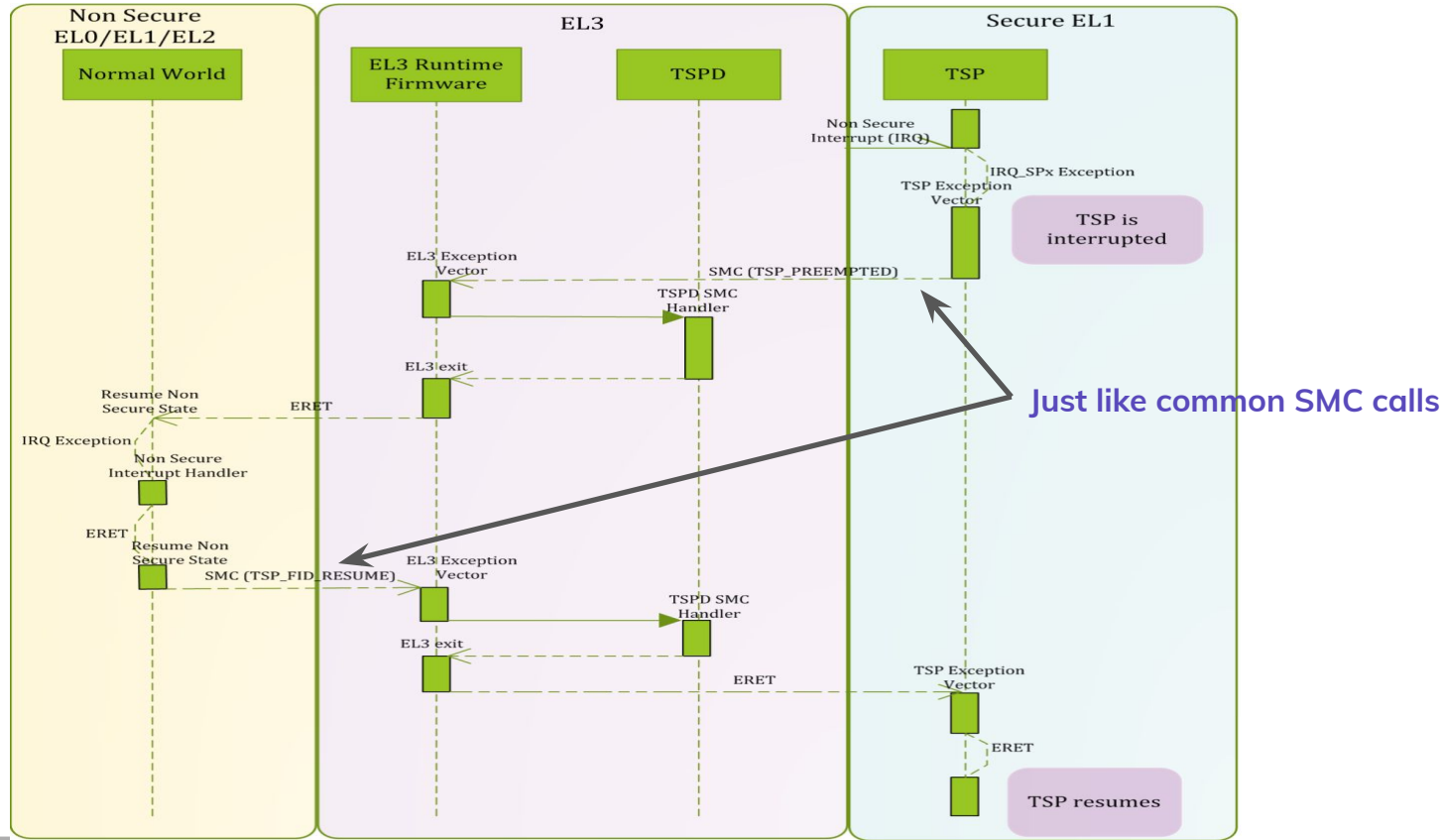
intr_type_desc[INTR_TYPE_NS].scr_el3[SECURE]     = 0x0
intr_type_desc[INTR_TYPE_NS].scr_el3[NOSECURE]   = 0x0
    
```



FIQ handling when running in normal world

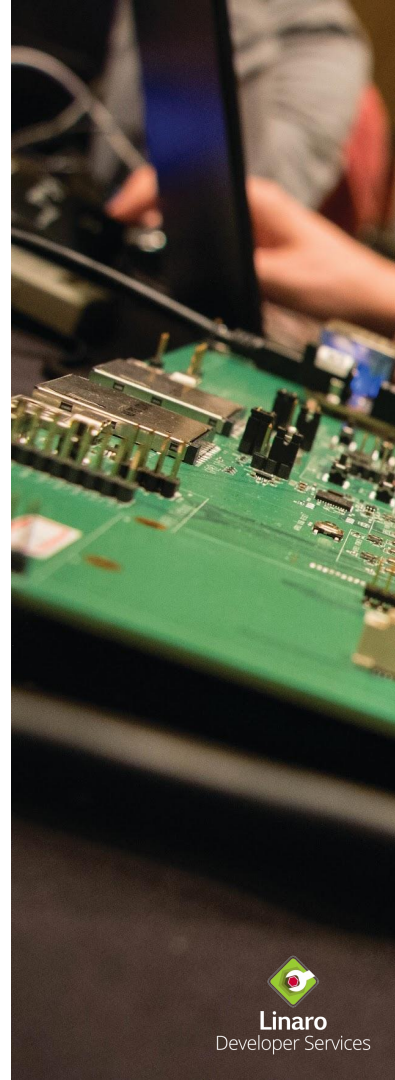


When IRQ happens in secure world



Secure/Realm world interfaces

- EL3 runtime interface (firmware)
 - Arm architecture services
 - Standard services
 - SiP/OEM specific services
- **EL3 runtime interface (Secure world)**
 - Secure-EL1 payload dispatcher
 - Interrupt handling
 - **FF-A standard protocol**
 - **Secure Partition Manager (SPM)**
- EL3 runtime interface (Realm world)
 - Realm Management Monitor (RMM)
 - Granule Protection Tables Library



FF-A standard protocol

The Armv8.4 architecture introduces the Virtualization extension in the Secure state (**Secure EL2**). This architectural features enable isolation of **mutually** mistrusting software components in the Secure state from each other.

Arm published [specification](#): Firmware Framework for Arm® A-profile processors (FF-A) which:

- Uses the Virtualization extension to **isolate** software images provided by an ecosystem of **vendors** from each other.
- Describes interfaces that **standardize** communication between the **various** software images.
 - Includes communication between images in the Secure world and Normal world.
- Generalizes **interaction** between a software image and privileged firmware in the Secure state.

Secure Partition Manager (SPM)

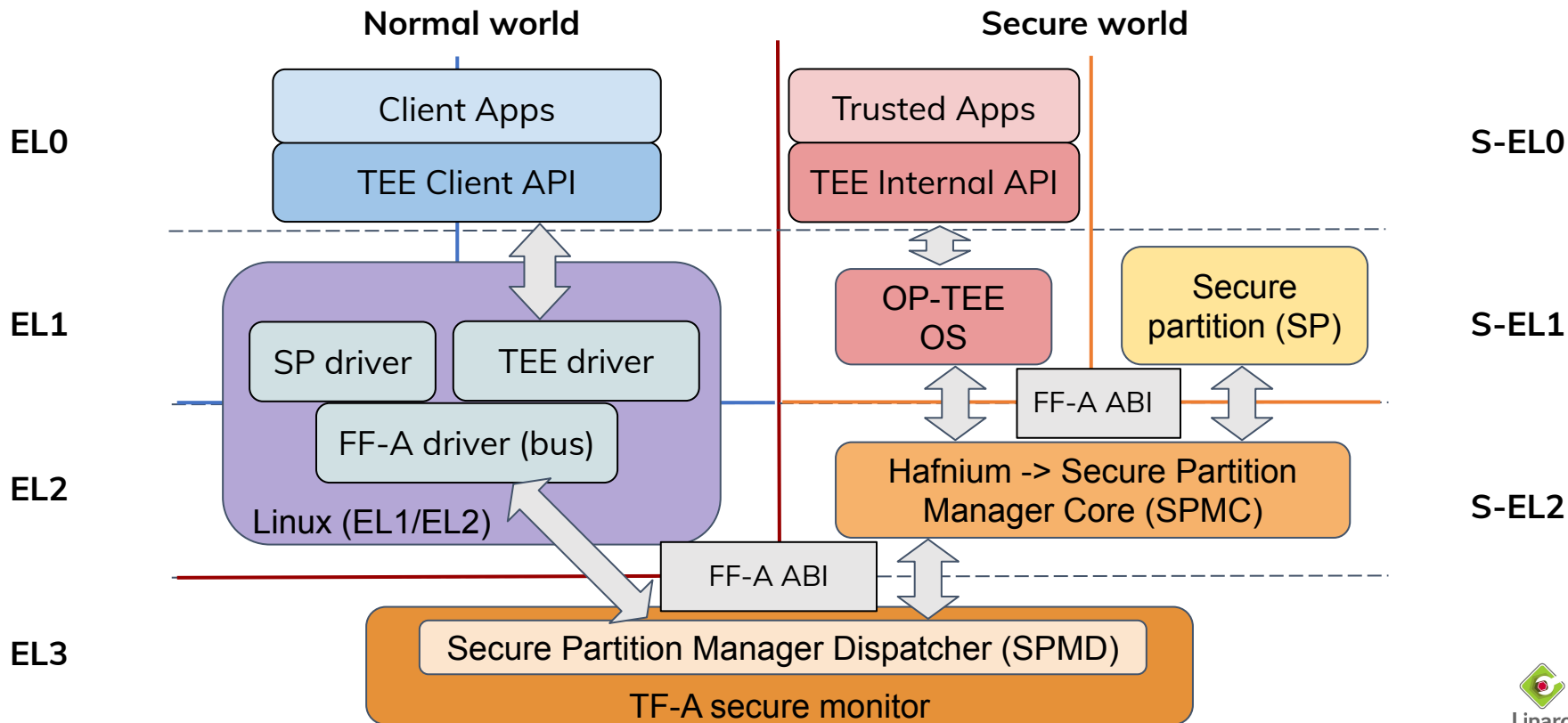
Post Armv8.4 architectures

With the advent of Secure world hypervisor (**S-EL2**), it is now possible to have **multiple Trusted OSes** executing and **isolated** from each other. The major use-case is to provide isolated trusted OS service corresponding to each **virtual machine** running in normal world.

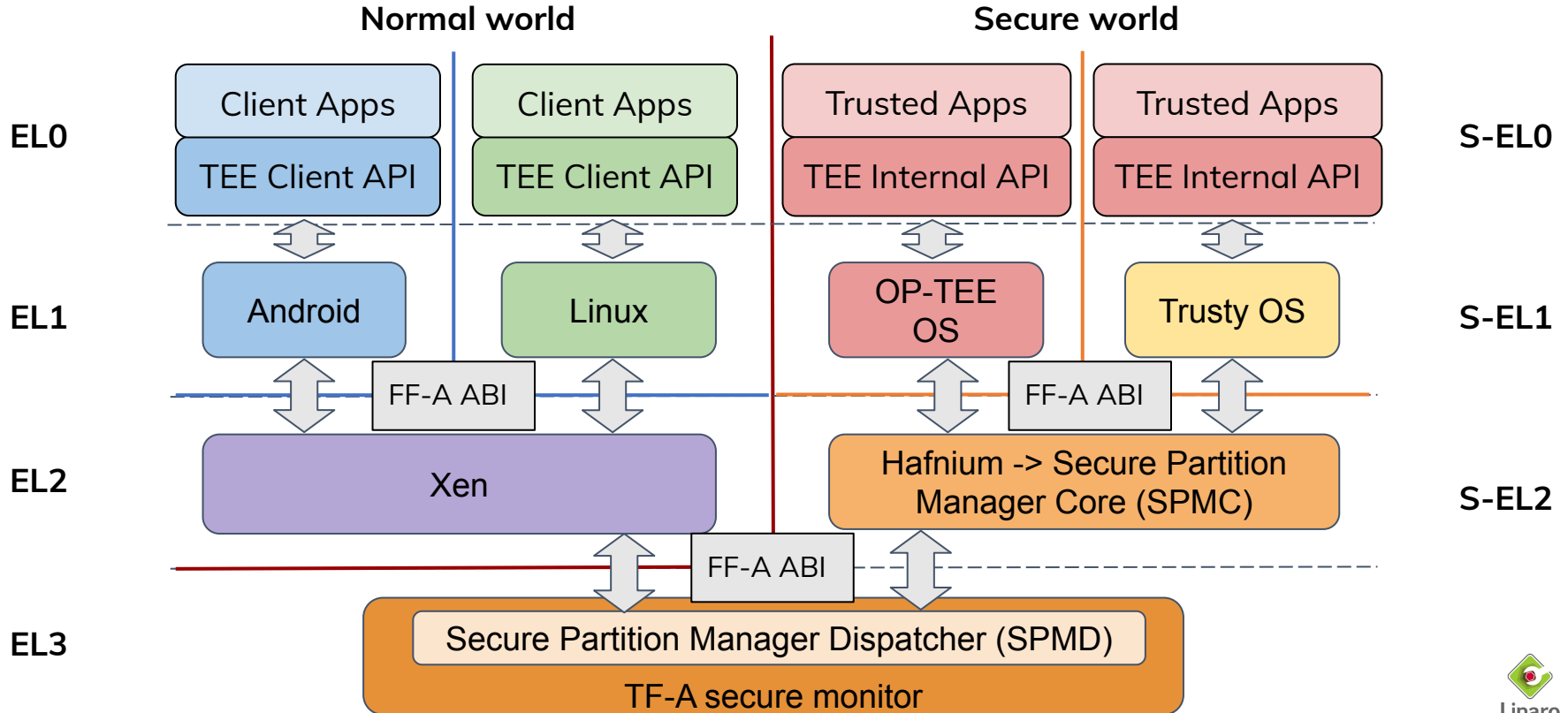
Secure hypervisor aka **Secure Partition Manager Core (SPMC)** runs at S-EL2. **Hafnium** is a reference SPMC implementation, a Trusted Firmware community project. EL3 runtime firmware provides a corresponding **Secure Partition Manager Dispatcher (SPMD)** responsible for:

- **Initializing** the SPMC.
- Routing **Firmware Framework-A (FF-A)** protocol messages among SPMC and REE hypervisor.
- SPMC specific **context management**.

TF-A: Armv8 (\geq v8.4) possible implementation

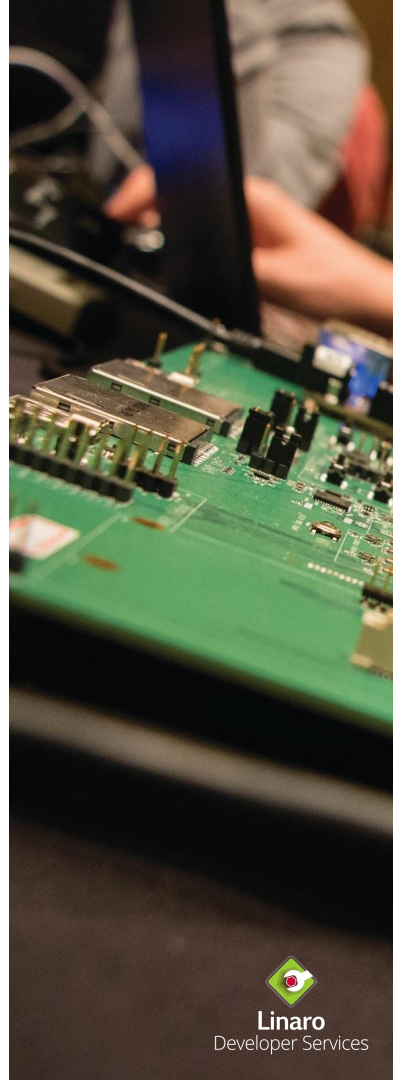


TF-A: Armv8 (>= v8.4) possible implementation



Secure/Realm world interfaces

- EL3 runtime interface (firmware)
 - Arm architecture services
 - Standard services
 - SiP/OEM specific services
- EL3 runtime interface (Secure world)
 - Secure-EL1 payload dispatcher
 - Interrupt handling
 - FF-A standard protocol
 - Secure Partition Manager (SPM)
- **EL3 runtime interface (Realm world)**
 - **Realm Management Monitor (RMM)**
 - **Granule Protection Tables Library**



EL3 runtime services

BL31 is the **most privileged** exception level where the **runtime resident** monitor firmware executes. TF-A provides **reference implementation** for EL3 monitor firmware.

Provide **runtime services**, such as:

- Arm architectural services
- Standard services such as **PSCI, SDEI, MM, TRNG** etc.
- SiP/OEM specific services
- Foundation to build:
 - Trusted Execution Environment (**TEE**)
 - **Realms** for Confidential Compute Architecture (**CCA**)

to lower exception levels (EL1/EL2 and S-EL1/S-EL2)

- **EL1** -> Rich OS or **EL2** -> Hypervisor
- **S-EL1** -> Trusted OS or **S-EL2** -> Secure hypervisor



Confidential Computing: Arm

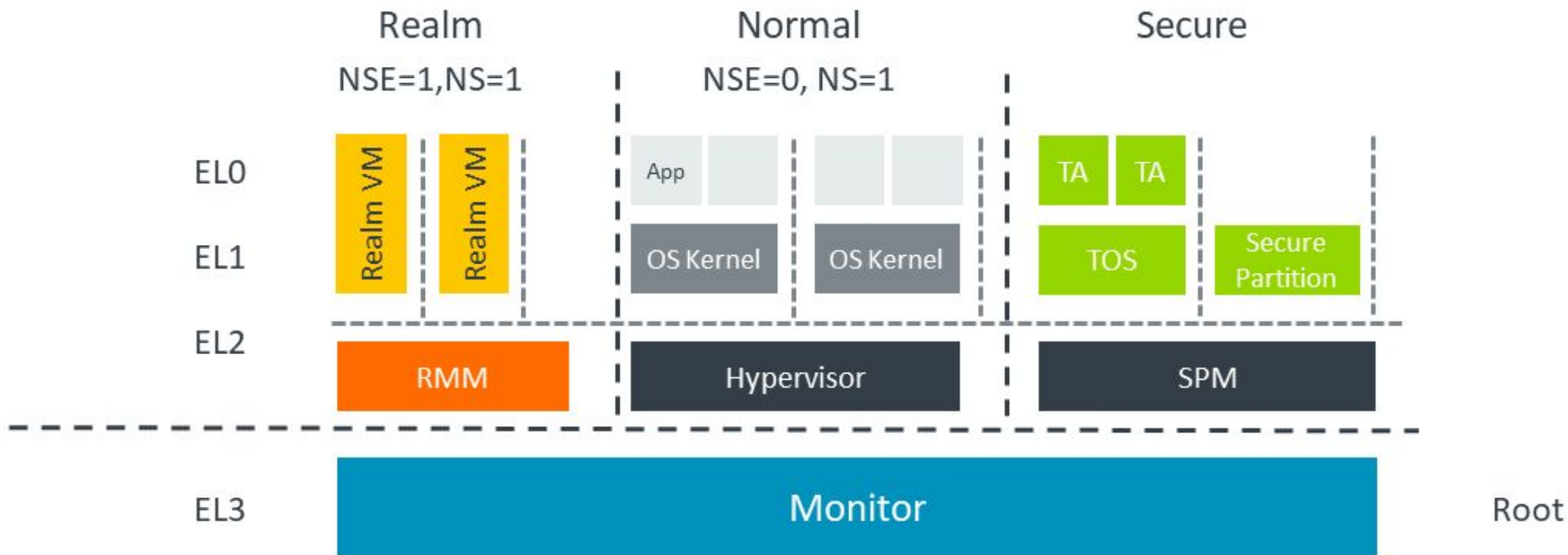
Confidential Computing is the protection of **data in use**, by performing computation within a **trustworthy hardware-backed** secure environment.

Although, **Arm TrustZone** can establish such an environment but there are **resource constraints** (like static allocation of DRAM, IOs etc.) when it comes to confidential computing requiring fully fledged **virtual machines** to run inside that secure environment.

With **Armv9 RME extension**, Arm introduced the concept of Realms. This allows normal world hypervisors to **manage resources** (DRAM, IOs etc.) **dynamically** but don't have **right to access** Realm code or data. It will in turn allows **confidentiality** of fully fledged virtual machine running as a Realm.



Arm CCA: Realms



Armv9 architecture: RME extension

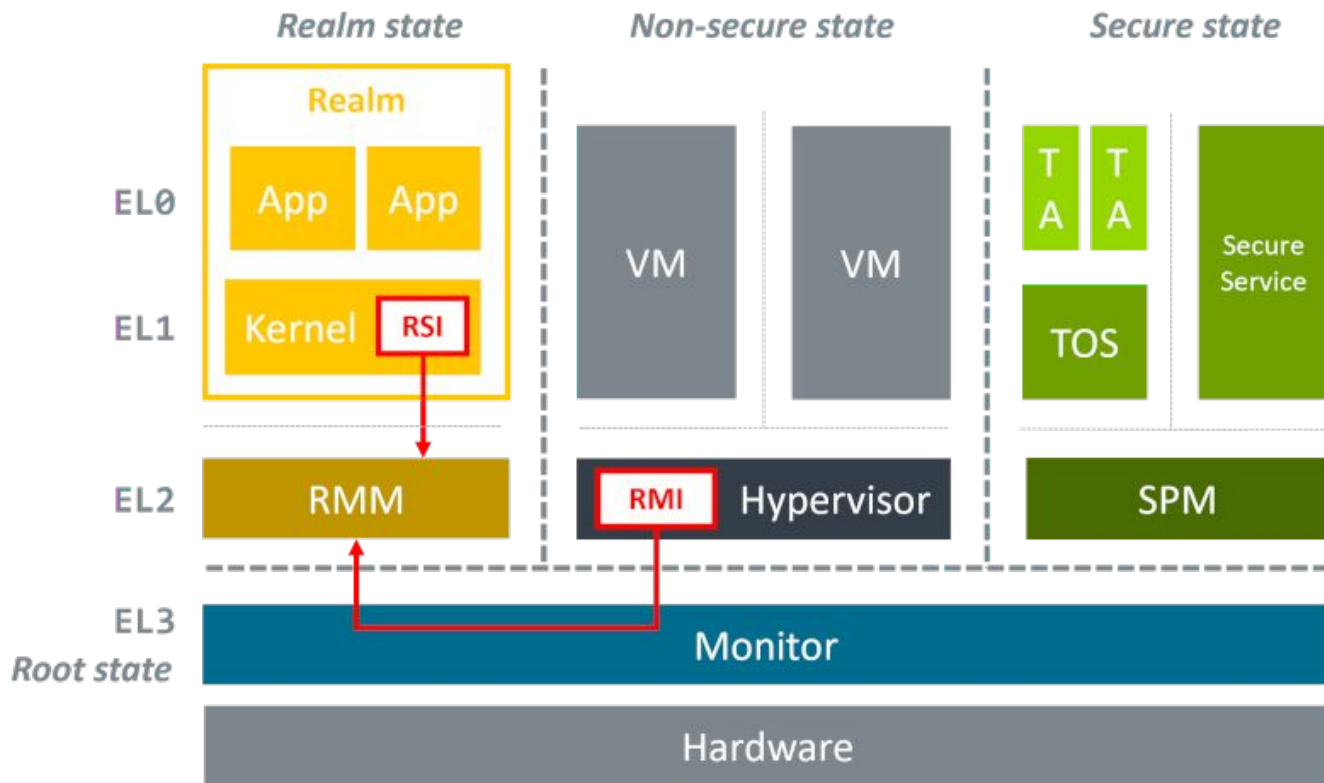
The Realm Management Extension (RME) supports a new class of **attestable isolation environment** called a **Realm**. Realms are **isolated** from the existing **normal and secure worlds** that we have today in TrustZone.

RME features:

- **Two additional** security states and address spaces: **Root** and **Realm**. TF-A BL31 runtime firmware runs in **Root world**. In the **Realm world**, a Realm Management Monitor firmware (**RMM**) manages the **execution** of Realm VMs and their **interaction** with the hypervisor.
- **Granule Protection Check (GPC)** feature to MMU downstream address translation.
- **Device assignment (DA)** in order for Realm to attest assigned device.
- **Memory Encryption Contexts (MEC)**, to allow for encrypted Realm, Secure, and Root physical address spaces, consumed by **Memory Protection Engine (MPE)**.



Realm VM execution



RMM: Realm Management Monitor firmware

The RMM is a software component that runs at **Realm EL2** and forms part of a system which implements the **Arm Confidential Compute Architecture** (Arm CCA).

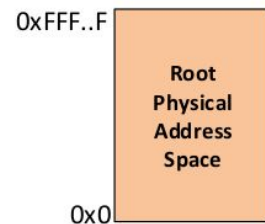
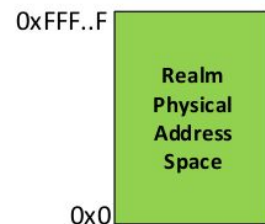
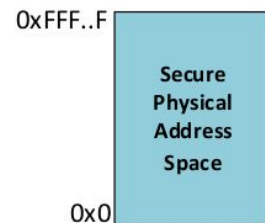
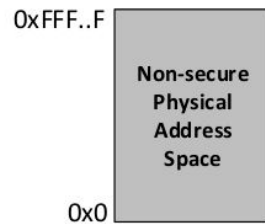
RMM provides a **Realm Management Interface (RMI)** towards normal world hypervisor to initiate and control realm VM. All the resource management (eg. memory) for realm VM happens via this interface. EL3 firmware provides **RMM dispatcher (RMMD)** to relay messages among RMM and normal world hypervisor.

RMM provides a **Realm Service Interface (RSI)** towards realm VM to request services such as:

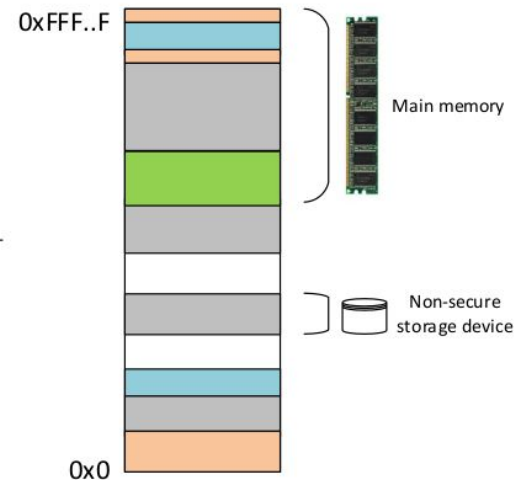
- Realm **attestation report** which also includes platform attestation.
- **Memory management** requests from the realm VM to the RMM.

RME Physical Address Spaces (PAS)

Architectoral Physical Address Spaces



Hardware Physical Address Space

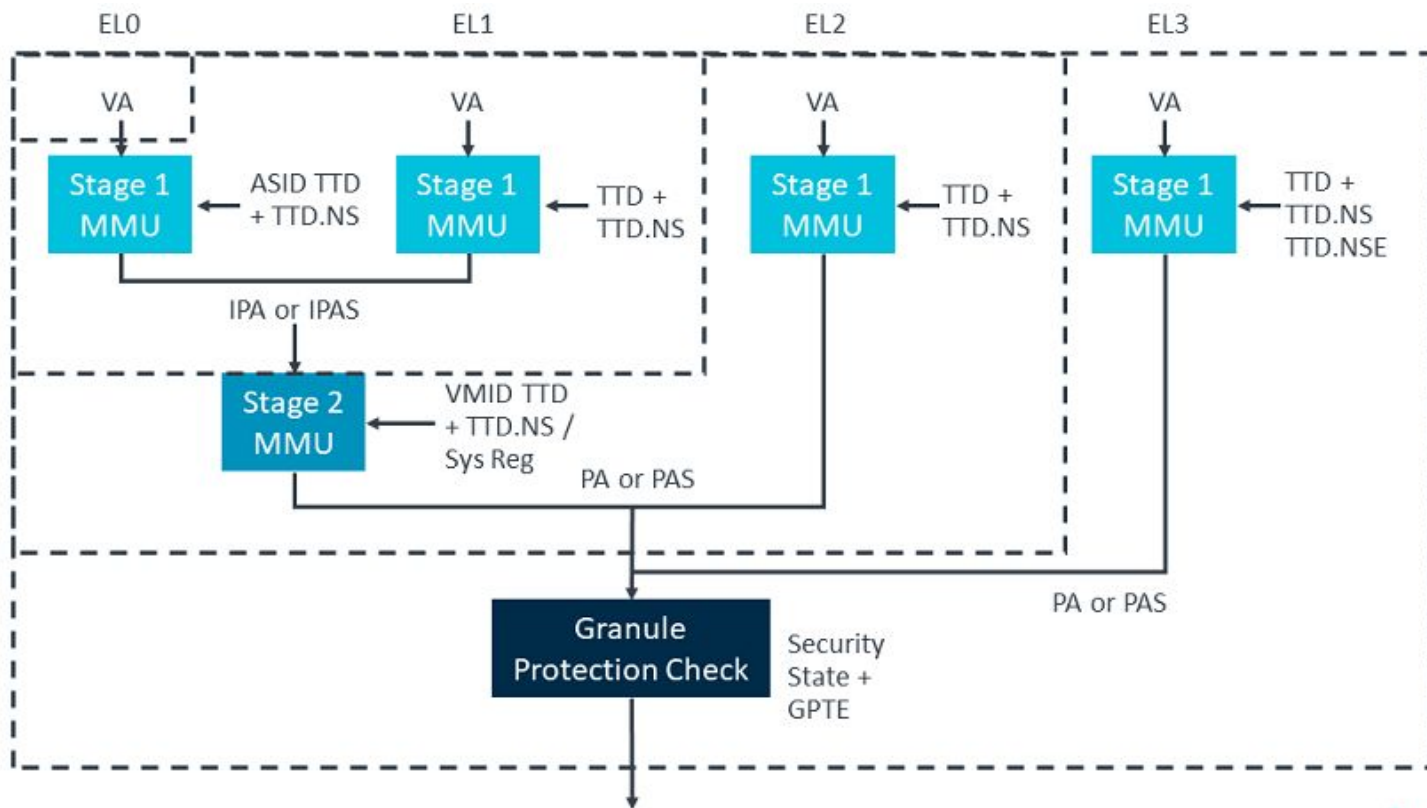


Memory isolation via architecturally-separate PAS

	Secure state	Non-secure state	Root state	Realm state
PAS[1:0]				
0b00 - Secure	Yes	No	Yes	No
0b01 - Non-secure	Yes	Yes	Yes	Yes
0b10 - Root	No	No	Yes	No
0b11 - Realm	No	No	Yes	Yes

PAS access table

Granule Protection Check (GPC)



Granule Protection Tables (GPT) library

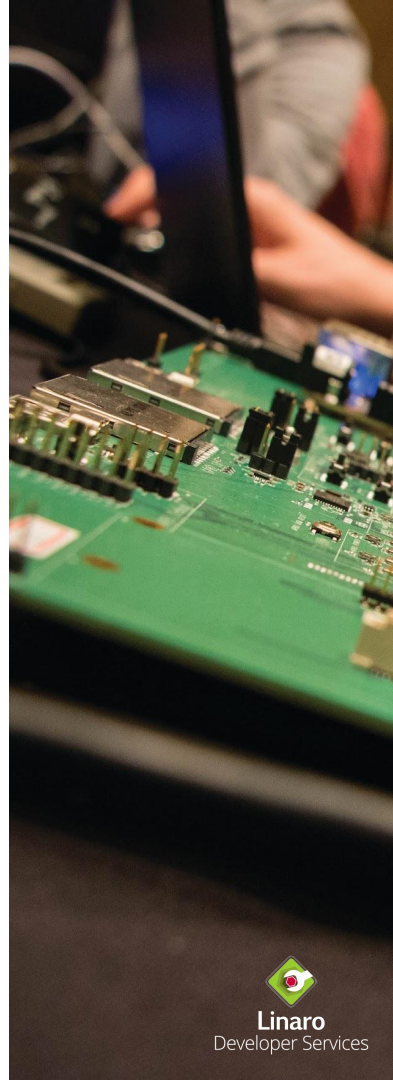
TF-A EL3 runtime firmware running in **Root world** is responsible for **initializing Granule Protection Tables (GPT)** to be **consumed by GPC** in order to enforce PAS memory isolation.

Role of EL3 wrt. PAS memory isolation:

- **Initialize** the GPTs based on a data structure containing information about the systems memory layout.
- **Configure** the system registers to enable granule protection checks based on these tables.
- **Transition** granules between different PAS (physical address spaces) **at runtime**.
- Supports upto 2 level GPT lookup
 - **Level 0**: large region in memory (block descriptor).
 - **Level 1**: relatively small regions (granules) of memory.



Lab sessions





Reminder: Preparation and boot

Preparatory steps remains similar as for TFA-01 lab sessions.

Boot cmdline:

```
qemu-system-aarch64 \  
  -machine virt,secure=on -cpu cortex-a57 \  
  -smp 4 -nographic -m 1G -bios flash.bin \  
  -drive \  
    file=./core-image-tfa-qemuarm64-secureboot.wic.qcow2,if=virtio,format=qcow2 \  
  -netdev user,id=eth0,hostfwd=tcp::2222-:22 \  
  -device virtio-net-device,netdev=eth0
```

Don't forget to source the
SDK environment setup
script!



LAB1 - Write and test a sample EL3 runtime service



Write EL3 SiP service for Qemu

In this lab session you need to write an **EL3 SiP service for Qemu** to return platform specific serial number.

Hints:

- Trainer has provided corresponding kernel module to test EL3 service here:
`$SDKTARGETSYSROOT/usr/src/lab-sessions/tfa0x-trusted_firmware_a/sdk/kernel_mod/`
- Modify TF-A source code to support Qemu SiP service, you can take reference from other platforms supporting SiP services like one [here](#).



```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/arm-smccc.h>
MODULE_LICENSE("GPL");
#define QEMU_SIP_SMC_GET_SERIAL_NUMBER \
    ARM_SMCCC_CALL_VAL(ARM_SMCCC_FAST_CALL, ARM_SMCCC_SMC_64, \
        ARM_SMCCC_OWNER_SIP, 1)
static int __init el3_service_init(void)
{
    struct arm_smccc_res res;
    printk(KERN_INFO "Hello, TF-A!\n");
    arm_smccc_1_1_invoke(QEMU_SIP_SMC_GET_SERIAL_NUMBER, &res);
    if (res.a0 == SMCCC_RET_SUCCESS)
        printk(KERN_INFO "TF-A EL3 service returned serial no: %08x\n", res.a1);
    else if (res.a0 == SMCCC_RET_NOT_SUPPORTED)
        printk(KERN_INFO "TF-A EL3 service not supported\n");
    return 0;
}
static void __exit el3_service_exit(void)
{
    printk(KERN_INFO "Goodbye, TF-A!\n");
}
module_init(el3_service_init);
module_exit(el3_service_exit);
```

EL3 service kernel module: Build

Build a kernel module to access EL3 service and use it to test your TF-A EL3 service for Qemu.

- Setup kernel source directory provided by SDK

```
$ cd $SDKTARGETSYSROOT/usr/src/kernel/
$ make scripts
$ make prepare
$ cd -
```
- Build the out-of-tree kernel module

```
$ cp -r $SDKTARGETSYSROOT/usr/src/lab-sessions/ .
$ cd lab-sessions/tfa0x-trusted_firmware_a/sdk/kernel_mod/
$ make
```
- Built EL3 service kernel module can be found as “el3_service.ko”.



EL3 service kernel module: Test

- Transfer “el3_service.ko” to your target (or you can use “scp” to copy binaries to your target)
- Install “el3_service.ko”

```
root@qemuarm64-secureboot:~# insmod ./el3_service.ko
```
- Explore the output console logs.



LAB2 - Enable Test Secure Payload (TSP) on Qemu



Enable Test Secure Payload (TSP) on Qemu

In this lab session you need to enable and test TSP and corresponding dispatcher (TSPD) on Qemu. For reference, you may refer to other TF-A platforms [\[1\]](#) [\[2\]](#) [\[3\]](#) supporting TSP.

Steps to build TF-A with TSP enabled for Qemu:

```
$ LDFLAGS= make CFLAGS= PLAT=qemu DEBUG=1 \  
    BL33=$SDKTARGETSYSROOT/boot/u-boot.bin BL32_RAM_LOCATION=tdram \  
    SPD=tspd all fip  
$ dd if=build/qemu/debug/bl1.bin of=flash-src.bin bs=4096 conv=notrunc  
$ dd if=build/qemu/debug/fip.bin of=flash-src.bin seek=64 bs=4096 conv=notrunc
```



LAB3 - Explore OP-TEE dispatcher flow



Build OP-TEE from source code

```
$ . /poky/sdk/path/environment-setup-cortexa57-poky-linux  
  
$ git clone https://github.com/OP-TEE/optee\_os.git  
$ cd optee_os  
$ git checkout -b 3.20.0 3.20.0  
$ make LDFLAGS= ARCH=arm CROSS_COMPILE64=aarch64-poky-linux- \  
    CFLAGS64=--sysroot=${SDKTARGETSYSROOT} \  
    PLATFORM=vexpress-qemu_armv8a CFG_ARM64_core=y \  
    ta-targets=ta_arm64 CFG_TEE_CORE_LOG_LEVEL=3
```

Build TF-A from source code

```
$ . /poky/sdk/path/environment-setup-cortexa57-poky-linux

$ git clone https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git
$ cd trusted-firmware-a
$ git checkout -b v2.9.0 v2.9.0

$ OPTEE_PATH=/optee-os/clone/path/
$ LDFLAGS= make CFLAGS= PLAT=qemu DEBUG=1 \
    BL32=$OPTEE_PATH/out/arm-plat-vexpress/core/tee-header_v2.bin \
    BL32_EXTRA1=$OPTEE_PATH/out/arm-plat-vexpress/core/tee-pager_v2.bin \
    BL32_EXTRA2=$OPTEE_PATH/out/arm-plat-vexpress/core/tee-pageable_v2.bin \
    BL33=$SDKTARGETSYSROOT/boot/u-boot.bin BL32_RAM_LOCATION=tdram \
    SPD=opteed all fip

$ dd if=build/qemu/debug/bl1.bin of=flash-src.bin bs=4096 conv=notrunc
$ dd if=build/qemu/debug/fip.bin of=flash-src.bin seek=64 bs=4096 conv=notrunc
```



Attach GDB during TF-A boot

Launch Qemu with cmdline:

```
qemu-system-aarch64 \  
-machine virt,secure=on -cpu cortex-a57 \  
-smp 2 -nographic -m 1G -bios trusted-firmware-a/flash-src.bin \  
-drive \  
  file=./core-image-tfa-qemuarm64-secureboot.wic.qcow2,if=virtio,format=qcow2 \  
-netdev user,id=eth0,hostfwd=tcp::2222-:22 \  
-device virtio-net-device,netdev=eth0 \  
-gdb tcp::1234 -S
```

“-gdb” option: Accept gdb connection over “tcp” port no. “1234”. QEMU defaults to starting the guest without waiting for gdb to connect; use “-S” too if you want it to not start execution.

“-S” option: Freeze CPU at startup (use 'c' to start execution).



Attach debugger during TF-A boot

Launch GDB in another terminal:

```
$ . /poky/sdk/path/environment-setup-cortexa57-poky-linux  
$ cd trusted-firmware-a  
$ aarch64-poky-linux-gdb
```

GDB setup commands:

```
(gdb) target remote localhost:1234  
(gdb) add-symbol-file build/qemu/debug/bl31/bl31.elf
```

Explore OP-TEE dispatcher: GDB breakpoints

Key OP-TEE dispatcher breakpoints:

```
(gdb) hbreak bl31_main
(gdb) hbreak runtime_svc_init
(gdb) hbreak
bl31_prepare_next_image_entry
(gdb) hbreak opteed_setup
(gdb) hbreak opteed_init_with_entry_point
(gdb) hbreak opteed_smc_handler
```

Boot up by writing “continue” in GDB:

```
(gdb) c <enter>
```

Use additional gdb commands to explore further.

```
print <expression>, info locals
backtrace
step, next, finish
list, display <expression>
```

If you step over something interesting and miss the details then you can always reboot, set a new breakpoint and take a closer look!



Thank you

support@linaro.org



Linaro
Developer Services