2023-04-11

Created: 2023-04-22 01:03:03

Chats

- OP-TEE.0327.md
  - `[08:58]` How is SMAF used in DRM?
  - `[08:59]` Is SMAF of OP-TEE?
  - `[10:00]` What is Clear Key?
  - `[18:46]` Which privilege levels do static TA and dynamic TA run in?
  - `[18:49]` Is a TA assigned to only one thread?
  - `[21:07]` Can I reset OP-TEE at run-time?
  - `[21:19]` Where does OP-TEE initialize secondary cores?
  - `[21:23]` Below code comes from part 1 of entry_a64.S of OP-TEE … …
  - `[21:24]` Below code comes from part 2 of entry_a64.S of OP-TEE … …
  - `[21:25]` Where to set up secondary cores in the given code?
  - `[21:48]` Where to reset primary core in the given code?
  - `[21:51]` What is the set_sp macro?
  - `[21:58]` When to enable secondary cores in OP-TEE?
  - `[22:08]` Please list functions and macros in entry_a64.S
  - `[22:10]` Please list functions and macros in entry_a64.S that I sent you before.
  - `[22:11]` Please list functions and macros in entry_a64.S that I give you before.
  - `[22:18]` Please list functions called in entry_a64.S that I give you before.
  - `[22:37]` What is psci?
  - `[22:40]` Who to call vector_cpu_on_entry?
  - `[22:42]` Who to call vector_cpu_on_entry with OP-TEE?
  - `[22:46]` What does vector_cpu_on_entry do with respect to OP-TEE?
  - `[23:10]` What are difference boot of OP-TEE between AArch64 and AArch32?
- TrustZone.0329.md
  - `[14:23]` Is HSM cost than TrustZone?
  - `[14:25]` Is HSM more performance than ARM TrustZone in crypto operations?
  - `[14:27]` What are security certifications of TrustZone?
- FF-A.0409.md
  - `[21:56]` When to enable secondary cores?
  - `[22:58]` Does TF-A support Arm32 or Arm64?
  - `[22:59]` Which types of ARM cores is supported by TF-A?
  - `[23:01]` Does Armv7-A has 64 bits?
  - `[23:05]` Must AArch64 have TF-A?
  - `[23:13]` Does Armv8-A has 32 bits?
  - `[23:23]` What is “Arm A-profile”?
  - `[23:24]` Does “Arm Firmware Framework for Arm A-profile” document is only about Cortex-A?
- Arm.0410.md
  - `[23:15]` What are types of ARM CPU? Please also describe them in a table with 32 bits and 64 bits.
  - `[23:18]` Does Armv8-A has 32 bits?
  - `[23:19]` Does Armv7-A has 64 bits?

Q & A

OP-TEE.0327.md

Question: `[08:58]` How is SMAF used in DRM?

Answer:

The Secure Memory Allocator Framework (SMAF) is a framework that can be used in conjunction with Digital Rights Management (DRM) systems to provide an additional layer of security for handling sensitive data. In a DRM context, sensitive data can include encryption keys, decrypted content, or any other data that must be protected from unauthorized access.

SMAF is designed to provide secure memory allocation and management in a system. It allocates memory from a secure memory pool that is reserved for trusted applications, such as DRM systems, and is inaccessible to the normal world or untrusted applications. This helps prevent unauthorized access, tampering, or copying of the sensitive data, ensuring that the content remains protected.

In a typical DRM implementation using SMAF:

1. A trusted application, such as a DRM module, requests secure memory allocation from the SMAF.
2. The SMAF allocates memory from the secure memory pool and provides a handle or reference to the trusted application.
3. The trusted application uses the allocated secure memory to store sensitive data, such as decryption keys or decrypted content.
4. The sensitive data remains protected within the secure memory, preventing unauthorized access from the normal world or untrusted applications.

By using SMAF, DRM systems can ensure that sensitive data is securely stored and managed, reducing the risk of unauthorized access, tampering, or copying of protected content.

Question: `[08:59]` Is SMAF of OP-TEE?

Answer:

Yes, SMAF is a part of the OP-TEE (Open Portable Trusted Execution Environment) framework. OP-TEE is an open-source implementation of the TEE (Trusted Execution Environment) for ARM processors, which provides a secure environment for running trusted applications.

SMAF, or Secure Memory Allocator Framework, is a component within the OP-TEE framework that enables the secure allocation and management of memory for trusted applications. It helps protect sensitive data, such as encryption keys and decrypted content, from unauthorized access in the normal world or by untrusted applications.

By integrating SMAF into OP-TEE, the framework can provide a higher level of security for applications, such as DRM systems, that require secure memory management for their sensitive data.

Question: `[10:00]` What is Clear Key?

Answer:

Clear Key is a simple, non-proprietary DRM (Digital Rights Management) system defined by the W3C (World Wide Web Consortium) as part of the Encrypted Media Extensions (EME) specification. EME is a standard interface for using DRM within web browsers to protect copyrighted content during streaming.

Clear Key is essentially a key management system that allows for the decryption of media content using clear-text (unencrypted) cryptographic keys. The primary purpose of Clear Key is to serve as a reference implementation for testing and development purposes. It is not meant to be a robust DRM solution for commercial use, as it lacks the sophisticated security features offered by other DRM systems, such as Google's Widevine or Microsoft's PlayReady.

In summary, Clear Key is a simple and open DRM system used mainly for testing and development of EME-supported web browsers and applications.

Question: `[18:46]` Which privilege levels do static TA and dynamic TA run in?

Answer:

In the context of ARM TrustZone and OP-TEE, there are two types of Trusted Applications (TAs): static and dynamic.

Static TAs are part of the OP-TEE OS itself, meaning they are compiled and linked together with the OP-TEE OS, and run at the same privilege level as the OP-TEE OS, which is Secure EL1 (Secure Exception Level 1) in the ARMv8 architecture.

Dynamic TAs, on the other hand, are standalone binaries that are loaded and executed by the OP-TEE OS on-demand. They run at a lower privilege level compared to the OP-TEE OS. In the ARMv8 architecture, dynamic TAs run at Secure EL0 (Secure Exception Level 0).

By running dynamic TAs at a lower privilege level, the system can provide better isolation between the TAs themselves and the OP-TEE OS, thereby improving security.

Question: `[18:49]` Is a TA assigned to only one thread?

Answer:

In OP-TEE, a Trusted Application (TA) can be single or multi-threaded, depending on its design and implementation. However, OP-TEE employs a single-threaded model by default, which means that each TA instance is typically associated with one thread, and there is no concurrency within a single instance of the TA.

If an application requires concurrent access to the TA, multiple instances of the TA can be created. Each instance runs in a separate thread, and the TA developer must take care of synchronization and shared resources within the TA code.

In summary, a TA is not inherently limited to a single thread, but the default behavior in OP-TEE is to use one thread per TA instance. If the TA is designed to support multiple threads, it can be done, but the developer must handle synchronization and shared resources carefully.

Question: `[21:07]` Can I reset OP-TEE at run-time?

Answer:

There isn't a direct mechanism to reset only the OP-TEE (Trusted Execution Environment) at runtime without affecting the rest of the system. OP-TEE runs alongside the Normal World operating system, and both share the same hardware resources.

Resetting the OP-TEE runtime would essentially require resetting the entire system, which would include the Normal World operating system (such as Linux). This is because OP-TEE interacts with the Normal World through shared memory and interrupt handling, and resetting OP-TEE without restarting the entire system may lead to undefined behavior or security vulnerabilities.

If you need to reset the state of a Trusted Application running within OP-TEE, you can unload the TA and reload it, essentially resetting the TA's state. However, this approach only resets the specific TA and not the entire OP-TEE environment.

Question: `[21:19]` Where does OP-TEE initialize secondary cores?

Answer:

OP-TEE initializes secondary cores during its boot process. The initialization happens in the `optee_os` codebase, which is the core component of OP-TEE. Specifically, the initialization can be found in the `core/arch/arm/kernel/` directory.

For ARM platforms, the file `core/arch/arm/kernel/entry_a32.S` (for 32-bit systems) or `core/arch/arm/kernel/entry_a64.S` (for 64-bit systems) contains the assembly code responsible for the secondary core initialization.

Secondary core initialization consists of the following steps:

1. Setup the core's stack pointer.
2. Enable the core's caches and MMU.
3. Initialize the core's GIC (Generic Interrupt Controller) interface.

4. Jump to the C function `__thread_std_smc_entry()` which handles the core's SMC (Secure Monitor Call) entry.

The actual boot process of the secondary cores is platform-specific and depends on the hardware and the bootloader used. In general, when the primary core starts the OP-TEE, it initializes the secondary cores, and they are put into a waiting state until they receive a wake-up event. The primary core is responsible for configuring and releasing the secondary cores from their waiting state.

Question: `[21:23]` Below code comes from part 1 of entry_a64.S of OP-TEE … …

Answer:

This code is the entry point of the OP-TEE operating system for ARM64-based systems. It is written in ARM64 assembly language and initializes the required hardware components and data structures before transferring control to the higher-level code in C.

Here's a summary of what the code does:

1. Saves input parameters such as the TOS FW config (if available) and the device tree address.
2. Sets the vector base address register to point to the reset vector table.
3. Sets system control register flags (sctlr_el1) to enable necessary features like instruction cache, stack alignment check, and WXN.
4. Initializes Memory Tagging Extension (MTE) and Pointer Authentication (PAUTH) if they are enabled and supported by the hardware.
5. Initializes the stack pointers for both secure (SP_EL0) and non-secure (SP_EL1) worlds.
6. Clears the .bss section in memory (zero-initialized global/static variables).
7. If CFG_NS_VIRTUALIZATION is enabled, clears the .nex_bss section in memory.
8. Sets up the thread_core_local data structure and enables aborts to handle exceptions.
9. Invalidates the data cache to make sure there are no stale cache entries.
10. Initializes the console for output.

Once the assembly code has completed these initializations, the execution is transferred to the higher-level C code, starting with `console_init()` function, which initializes the console for OP-TEE's debug output. This code essentially prepares the environment for OP-TEE to execute and provides a solid foundation for the rest of the software stack.

Question: `[21:24]` Below code comes from part 2 of entry_a64.S of OP-TEE … …

Answer:

The provided code is from part 2 of entry_a64.S of OP-TEE (Open Portable Trusted Execution Environment). OP-TEE is an open-source implementation of the Trusted Execution Environment (TEE) with a client API for communication between the client application running in the normal world and the trusted application running in the secure world.

This part of the code deals with various configurations and initializations related to the execution environment. Some of the functionalities include:

1. Initializing memory tagging if the feature is enabled (CFG_MEMTAG).
2. Initializing the MMU (Memory Management Unit) with or without Address Space Layout Randomization (ASLR) enabled (CFG_CORE_ASLR).
3. Reinitializing the console in case of ASLR.
4. Initializing partition tables for each partition in case of virtualization (CFG_NS_VIRTUALIZATION).
5. Initializing the primary core and setting up the secondary cores.
6. Initializing per-core features like memory tagging and pointer authentication (CFG_MEMTAG, CFG_CORE_PAUTH).
7. Enabling the MMU and updating the vector table.
8. Handling CPU on events for secondary cores (cpu_on_handler).
9. Handling unhandled CPU events (unhandled_cpu).

The code also contains various assembly macros and functions for handling different exception types and initializing/resetting the vector table (reset_vect_table).

Question: `[21:25]` Where to set up secondary cores in the given code?

Answer:

In the given code, the secondary cores setup is handled by the `cpu_on_handler` function. This function is responsible for preparing the secondary core when it is powered on. Here's an overview of the `cpu_on_handler` function:

```
FUNC cpu_on_handler , :
    mov x19, x0
    mov x20, x1
    mov x21, x30

    adr x0, reset_vect_table
    msr vbar_el1, x0
    isb

    set_sctlr_el1
    isb

    /* Enable aborts now that we can receive exceptions */
    msr daifclr, #DAIFBIT_ABT

    bl  __get_core_pos
    bl  enable_mmu

    /* Setup SP_EL0 and SP_EL1, SP will be set to SP_EL0 */
    set_sp

#ifdef CFG_MEMTAG
    init_memtag_per_cpu
#endif
#ifdef CFG_CORE_PAUTH
    init_pauth_per_cpu
#endif

    mov x0, x19
    mov x1, x20
#ifdef CFG_CORE_FFA
    bl  boot_cpu_on_handler
    b   thread_ffa_msg_wait
#else
    mov x30, x21
    b   boot_cpu_on_handler
#endif
END_FUNC cpu_on_handler
```

The `cpu_on_handler` function does the following:

1. Sets up the exception vector table by loading the address of `reset_vect_table` into the `vbar_el1` register.
2. Configures the System Control Register (`sctlr_el1`) and enables aborts to receive exceptions.
3. Gets the core position and enables the MMU.
4. Sets up the stack pointers for the current Exception Level (EL) by calling `set_sp`.
5. Initializes per-core features like memory tagging and pointer authentication if they are enabled (CFG_MEMTAG, CFG_CORE_PAUTH).
6. Calls the `boot_cpu_on_handler` function to complete the setup for the secondary core.

After the setup is complete, the secondary core will either wait for a Fast Function call (in case of CFG_CORE_FFA) or continue with the execution of the `boot_cpu_on_handler` function.

Question: `[21:48]` Where to reset primary core in the given code?

Answer:

In the given code, there's no specific mention of a function or a code block that explicitly resets the primary core. However, if

you need to reset the primary core, you can create a custom function that will perform the necessary actions, such as disabling interrupts and performing a soft reset.

You could create a function like this:

```
void reset_primary_core(void)
{
    // Disable interrupts
    local_irq_disable();
    local_fiq_disable();

    // Perform a soft reset
    uintptr_t reset_address = 0; // Replace 0 with the address of the reset handler for your system
    void (*reset_handler)(void) = (void (*)(void))reset_address;
    reset_handler();
}
```

Keep in mind that the exact process for resetting the primary core depends on your system's architecture and hardware. The example above is just a basic idea of how you could approach resetting the primary core. You will need to consult your system's documentation and adapt the code accordingly.

Question: `[21:51]` What is the set_sp macro?

Answer:

In the provided code, `set_sp` is a macro mentioned in the comments, but it is not defined in the given code. It is likely that the macro is used to set up the stack pointers for different Exception Levels (EL) in the ARM architecture. Setting up the stack pointers is an essential step during the initialization of a processor core.

In ARMv8, there are separate stack pointers for different ELs, and they have different names depending on the current EL:

- SP_EL0: Stack pointer for Exception Level 0
- SP_EL1: Stack pointer for Exception Level 1
- SP_EL2: Stack pointer for Exception Level 2
- SP_EL3: Stack pointer for Exception Level 3

The `set_sp` macro would be responsible for initializing these stack pointers for the current core. Depending on the implementation, it might involve:

1. Setting up stack pointer for the current EL (e.g., SP_EL1)
2. Setting up stack pointer for EL0 (SP_EL0)

Here's an example of what the `set_sp` macro might look like:

```
.macro set_sp
    // Set up the stack pointer for the current EL
    ldr x0, =__stack_start
    mov sp, x0

    // Set up the stack pointer for EL0 (SP_EL0)
    ldr x0, =__stack_el0_start
    msr sp_el0, x0
.endm
```

In this example, `__stack_start` and `__stack_el0_start` are symbols representing the starting addresses of the stack memory regions for the current EL and EL0, respectively. The macro initializes the stack pointers using these addresses.

Keep in mind that this is just a simple example of what `set_sp` macro might look like, and the actual implementation may differ based on your system requirements and the specific ARMv8-A processor variant used.

Question: `[21:58]` When to enable secondary cores in OP-TEE?

Answer:

In OP-TEE (Open Portable Trusted Execution Environment), secondary cores are typically enabled during the boot process, after the primary core has completed initialization tasks, including setting up secure memory, initializing OP-TEE, and launching the Rich Execution Environment (REE).

To enable secondary cores in OP-TEE, follow these steps:

1. Primary core initialization: The primary core initializes OP-TEE and other necessary system components. It sets up the secure memory, initializes the OP-TEE OS, and starts the REE (e.g., Linux kernel).
2. Secondary core synchronization: The primary core waits for secondary cores to reach a synchronization point, usually through a spin loop. Secondary cores wait for a signal from the primary core to continue the boot process.
3. Signal secondary cores: Once the primary core finishes the initialization process, it signals the secondary cores to continue the boot process. This can be done by writing a value to a shared memory location or using hardware-specific mechanisms.
4. Secondary core initialization: Secondary cores initialize their local data structures and set up their own exception handlers. They also initialize their stack pointers and register with the OP-TEE OS.
5. Secondary cores enter OP-TEE: Secondary cores enter the OP-TEE OS and start executing secure tasks.

The exact implementation details of enabling secondary cores may vary depending on the platform, hardware, and OP-TEE version. The steps mentioned above provide a general guideline for the process. To understand the specifics for your platform, refer to the platform-specific documentation and code available in the OP-TEE source tree.

Question: `[22:08]` Please list functions and macros in entry_a64.S

Answer:

`entry_a64.S` is an assembly file that provides low-level entry points for the AArch64 architecture in the OP-TEE OS. Here's a list of functions and macros typically found in the `entry_a64.S` file:

Functions: 1. `reset_primary_core` : Handles the primary core's reset and initialization. 2. `reset_secondary_core` : Handles the secondary core's reset and initialization. 3. `ns_entry` : Handles transitions from the Normal World to the Secure World. 4. `smc_handler` : Handles SMC (Secure Monitor Call) instructions. 5. `fiq_handler` : Handles FIQ (Fast Interrupt Request) exceptions. 6. `irq_handler` : Handles IRQ (Interrupt Request) exceptions. 7. `sync_abort_handler` : Handles synchronous abort exceptions. 8. `prefetch_abort_handler` : Handles prefetch abort exceptions. 9. `data_abort_handler` : Handles data abort exceptions. 10. `undef_handler` : Handles undefined instruction exceptions. 11. `svc_handler` : Handles SVC (Supervisor Call) exceptions.

Macros: 1. `SAVE_GP_REGS` : Saves general-purpose registers to the stack. 2. `RESTORE_GP_REGS` : Restores general-purpose registers from the stack. 3. `SAVE_SP_REGS` : Saves stack pointer registers to the stack. 4. `RESTORE_SP_REGS` : Restores stack pointer registers from the stack. 5. `SAVE_SYS_REGS` : Saves system registers to the stack. 6. `RESTORE_SYS_REGS` : Restores system registers from the stack. 7. `SAVE_FPU_REGS` : Saves floating-point registers to the stack. 8. `RESTORE_FPU_REGS` : Restores floating-point registers from the stack.

Please note that the actual functions and macros in your specific version of the OP-TEE OS might differ. Make sure to consult the source code of the version you're working with for an accurate list.

Question: `[22:10]` Please list functions and macros in entry_a64.S that I sent you before.

Answer:

My apologies for the confusion earlier, as I am an AI language model, I am unable to access external files or content that you've sent. Please share the relevant parts of the code, and I'd be happy to help with your question.

Question: `[22:11]` Please list functions and macros in entry_a64.S that I give you before.

Answer:

Based on the code you provided earlier, here is a list of functions and macros in entry_a64.S:

Functions: 1. _start 2. relocate (if CFG_CORE_ASLR is defined) 3. enable_mmu 4. cpu_on_handler 5. unhandled_cpu 6. reset_vect_table

Macros: 1. check_vector_size 2. BTI (if defined)

If you have any questions or need further information about these functions or macros, please let me know!

Question: `[22:18]` Please list functions called in entry_a64.S that I give you before.

Answer:

Here is a list of functions called in the entry_a64.S code you provided earlier:

1. boot_init_memtag (if CFG_MEMTAG is defined)
2. get_aslr_seed (if CFG_CORE_ASLR is defined)
3. core_init_mmu_map
4. __get_core_pos
5. enable_mmu
6. console_init (if CFG_CORE_ASLR is defined)
7. core_mmu_set_default_prtn_tbl (if CFG_NS_VIRTUALIZATION is defined)
8. boot_init_primary_early
9. init_memtag_per_cpu (if CFG_MEMTAG is defined)
10. thread_get_core_local
11. boot_init_primary_late
12. init_pauth_per_cpu (if CFG_CORE_PAUTH is defined)
13. plat_get_random_stack_canary (if _CFG_CORE_STACK_PROTECTOR is defined)
14. dcache_cleaninv_range
15. thread_clr_boot_thread (if CFG_NS_VIRTUALIZATION is not defined)
16. thread_spmc_register_secondary_ep (if CFG_CORE_FFA is defined)
17. thread_ffa_msg_wait (if CFG_CORE_FFA is defined)
18. boot_cpu_on_handler (if CFG_CORE_FFA is not defined)
19. panic_at_smc_return (if CFG_CORE_FFA is not defined)

These are the functions called in the provided entry_a64.S file.

Question: `[22:37]` What is psci?

Answer:

PSCI (Power State Coordination Interface) is a standardized interface for power management in ARM-based systems, specifically targeting multicore and multiprocessor SoCs (System on a Chip). The PSCI specification is developed and maintained by ARM and provides a common API for software running on ARM-based systems to control power states of CPUs and the system as a whole.

PSCI enables software layers, such as operating systems and hypervisors, to manage the power state of individual CPU cores and clusters of cores, as well as to perform system-level power management tasks, such as system suspend and resume. The PSCI interface is typically implemented in firmware and is accessible to the software layers through a defined set of SMC (Secure Monitor Call) instructions.

Some common PSCI functions include:

1. CPU_ON: Used to power on a specific CPU core.
2. CPU_OFF: Powers off the calling CPU core.
3. CPU_SUSPEND: Suspends the calling CPU core.
4. CPU_RESUME: Resumes a suspended CPU core.

5. SYSTEM_SUSPEND: Suspends the entire system.
6. SYSTEM_RESET: Performs a system reset.
7. SYSTEM_OFF: Powers off the entire system.

PSCI allows for better coordination of power management across different cores and system components, which is essential for efficient energy usage and optimized performance in modern ARM-based systems.

Question: `[22:40]` Who to call vector_cpu_on_entry?

Answer:

`vector_cpu_on_entry` is a function that is typically called when a CPU core is powered on in the context of an ARM-based system using PSCI for power management. The function is usually defined in the low-level firmware, such as ARM Trusted Firmware (ATF) or U-Boot.

To call `vector_cpu_on_entry`, you will need to use the PSCI interface. The `CPU_ON` PSCI function is used to power on a specific CPU core, and it's usually called by the operating system or another higher-level software component that manages power states of the CPU cores.

Here's an example of how you might call the `CPU_ON` function using PSCI:

1. Prepare the necessary arguments, such as the target CPU's MPIDR (Multiprocessor Affinity Register) and the entry point address of the `vector_cpu_on_entry` function.
2. Issue an SMC (Secure Monitor Call) instruction with the appropriate PSCI function ID for `CPU_ON` and the prepared arguments.

Please note that the actual implementation details will depend on your specific system configuration and the software components you are using. The operating system, hypervisor, or other system software components usually take care of calling the PSCI `CPU_ON` function and setting up the necessary entry points. As a developer, you typically don't need to call the `vector_cpu_on_entry` function directly.

Question: `[22:42]` Who to call vector_cpu_on_entry with OP-TEE?

Answer:

In the context of OP-TEE, the `vector_cpu_on_entry` function is not used directly. OP-TEE relies on ARM Trusted Firmware (ATF) for power management, which handles calling the appropriate entry points when powering on CPU cores.

When using OP-TEE and ATF, the ATF will handle the PSCI calls, such as `CPU_ON`, and call the appropriate entry points for the non-secure world (e.g., the operating system) and the secure world (e.g., OP-TEE). In this case, you do not call `vector_cpu_on_entry` directly.

If you need to bring up secondary CPU cores and start executing OP-TEE on them, you should follow these steps:

1. Make sure OP-TEE is built with secondary core support enabled. This is usually done by defining `CFG_TEE_CORE_NB_CORE` with the number of cores you want to support.
2. In the non-secure world (e.g., Linux kernel), use the PSCI interface to power on the secondary cores. You can do this by calling `cpu_up()` or similar functions depending on your kernel configuration.

When the secondary core is powered on, the ATF will handle the rest, setting up the appropriate entry points for both non-secure and secure worlds. The secondary core will then start executing OP-TEE in the secure world, and the non-secure world operating system on the other side.

Question: `[22:46]` What does vector_cpu_on_entry do with respect to OP-TEE?

Answer:

`vector_cpu_on_entry` is not a specific function in OP-TEE. However, I believe you are referring to a generic concept of an entry point for secondary cores in a multi-core system.

In OP-TEE, the entry point for secondary cores is defined in `core/arch/arm/kernel/entry_a64.S` (for ARMv8-A) or `core/arch/arm/kernel/entry_a32.S` (for ARMv7-A) depending on the architecture. The entry point function is named `__secondary_init` for both ARMv8-A and ARMv7-A architectures.

When a secondary core is powered on in a multi-core system, the ATF (ARM Trusted Firmware) handles the process and sets up the appropriate entry points for both the non-secure world (e.g., the operating system) and the secure world (e.g., OP-TEE). The secondary core then starts executing the OP-TEE code at the `__secondary_init` entry point in the secure world.

The `__secondary_init` function performs the following tasks:

1. Initialize the secondary core's CPU registers.
2. Set up the MMU and configure the memory map.
3. Call `generic_entry_a64` (for ARMv8-A) or `generic_entry_a32` (for ARMv7-A) to perform generic initialization, such as setting up the stack, enabling interrupts, and initializing the core's private GIC (Generic Interrupt Controller) interface.
4. Call `thread_init_per_cpu` to initialize the per-CPU data structures.
5. Enter the main scheduler loop by calling `thread_std_smc_entry` (for ARMv8-A) or `thread_std_svc_entry` (for ARMv7-A).

After these steps, the secondary core is ready to execute secure tasks in OP-TEE along with the primary core.

Question: `[23:10]` What are difference boot of OP-TEE between AArch64 and AArch32?

Answer:

OP-TEE supports both AArch64 (ARMv8-A) and AArch32 (ARMv7-A) architectures. While the overall boot process is similar for both, there are some differences due to the architectural distinctions. Here are the main differences in the boot process of OP-TEE between AArch64 and AArch32:

1. Entry point: The entry point for the primary core is different for AArch64 and AArch32. In AArch64, it's defined in `core/arch/arm/kernel/entry_a64.S`, while in AArch32, it's defined in `core/arch/arm/kernel/entry_a32.S`.
2. Register usage: AArch64 and AArch32 have different register usage conventions. AArch64 has 31 general-purpose registers (X0-X30), while AArch32 has 16 general-purpose registers (R0-R15). The register naming and usage in the assembly code for these architectures will differ.
3. MMU setup: AArch64 uses a different memory translation scheme compared to AArch32. Consequently, the MMU configuration code in `core/arch/arm/mm/core_mmu_lpae.c` (for AArch64) and `core/arch/arm/mm/core_mmu_v7.c` (for AArch32) differ in setting up the translation tables and configuring the memory attributes.
4. Exception levels: AArch64 introduces Exception Levels (EL0-EL3) that define different privilege levels for software execution. AArch32 uses the traditional ARM privilege levels (User, FIQ, IRQ, Supervisor, Abort, and Undefined) instead. The exception handling and context switching code will differ between the two architectures due to this distinction.
5. Interrupt handling: AArch64 uses the GICv3 interrupt controller, while AArch32 typically uses the GICv2 interrupt controller. The initialization and handling of interrupts will differ between the two architectures due to the differences in GIC versions.
6. Function calling convention: AArch64 and AArch32 have different function calling conventions. For example, AArch64 passes the first eight arguments in registers X0-X7, while AArch32 passes arguments using registers R0-R3 and the stack. The assembly code for function calls and stack frame setup will differ between the two architectures.

Despite these differences, the high-level boot process and functionality of OP-TEE remain consistent between AArch64 and AArch32. The architectural differences are primarily reflected in the low-level implementation details, such as register usage, memory management, and exception handling.

TrustZone.0329.md

Question: `[14:23]` Is HSM cost than TrustZone?

Answer:

It's difficult to make a direct comparison between the cost of a hardware security module (HSM) and TrustZone, as they serve different purposes and have different implementations. Here are some points to consider:

1. HSM: A standalone, tamper-resistant hardware device that provides secure key storage, cryptographic processing, and other security functions. HSMs are typically used in high-security environments, like financial institutions, where the protection of sensitive information is crucial. They can be relatively expensive due to their robust security features and the fact that they are dedicated hardware devices.
2. TrustZone: A hardware-based security feature in Arm processors that provides isolation between the Secure World and the Normal World. TrustZone is integrated into Arm processors, so the cost is essentially built into the processor itself. TrustZone may provide a lower level of security compared to an HSM, but it's generally more cost-effective since it's already a part of the processor.

In summary, HSMs tend to be more expensive than TrustZone because they are dedicated, standalone hardware devices with more robust security features. TrustZone, on the other hand, is a built-in security feature in Arm processors and is generally more cost-effective. However, the level of security provided by TrustZone may not be as high as that of an HSM. The choice between the two will depend on the specific security requirements and budget of a given project or organization.

Question: `[14:25]` Is HSM more performance than ARM TrustZone in crypto operations?

Answer:

In general, hardware security modules (HSMs) are designed to be more performant in cryptographic operations compared to ARM TrustZone. This is due to several reasons:

1. Dedicated hardware: HSMs are standalone devices specifically designed for cryptographic operations, including secure key storage, encryption, decryption, and signing. They have dedicated cryptographic accelerators and coprocessors optimized for these tasks, leading to better performance.
2. Reduced overhead: TrustZone, while providing a secure environment for cryptographic operations, still shares the same processor with the normal world. This sharing can lead to resource contention and increased overhead, potentially affecting performance.
3. Tamper resistance: HSMs are designed with robust tamper resistance in mind, including active measures to prevent unauthorized access or tampering. TrustZone, while providing a level of isolation between the secure and normal worlds, may not offer the same level of physical tamper resistance.
4. Scalability: HSMs can be used in a cluster or load-balanced configuration to handle large-scale cryptographic operations, providing better performance and throughput.

However, it's important to note that the specific performance difference between HSMs and TrustZone in cryptographic operations will depend on the implementation details, workload, and hardware configuration. For some use cases, TrustZone may provide sufficient performance, while in other scenarios, an HSM may be necessary to meet strict performance and security requirements.

Question: `[14:27]` What are security certifications of TrustZone?

Answer:

ARM TrustZone technology provides a foundation for building secure systems, but the security certifications depend on the specific implementations and products built on top of it. Different implementations may undergo different security evaluation processes and achieve various certifications based on their design, features, and target markets.

Some common security certifications that TrustZone-based products can achieve include:

1. Common Criteria (CC): TrustZone-based implementations can be certified under Common Criteria, an international standard (ISO/IEC 15408) for computer security certification. The level of assurance for a specific product is determined by its Evaluation Assurance Level (EAL), ranging from EAL1 (lowest) to EAL7 (highest).
2. FIPS 140-2/140-3: TrustZone-based products can also be certified under the Federal Information Processing Standards (FIPS) 140-2 or 140-3, which are U.S. government security standards for cryptographic modules. These standards have various security levels, with Level 1 providing the lowest security and Level 4 providing the highest security.

3. GlobalPlatform: Some TrustZone implementations, particularly those involving secure elements or Trusted Execution Environments (TEEs) such as OP-TEE, can be certified by GlobalPlatform. GlobalPlatform sets security requirements and evaluates products for secure elements, TEEs, and other secure components.
4. Payment Card Industry (PCI): TrustZone-based implementations targeting payment systems may achieve PCI compliance, ensuring that the product meets the security standards for processing, storing, or transmitting credit card information.

Note that achieving these certifications depends on the specific product, its implementation, and the security features provided. It's essential to verify the certifications of a TrustZone-based product depending on the use case and the required security level.

FF-A.0409.md

Question: `[21:56]` When to enable secondary cores?

Answer:

Secondary cores are typically enabled after the primary core has successfully initialized the system and started the operating system. The exact process for enabling secondary cores can vary depending on the hardware and operating system being used. Here is a general outline of when secondary cores are enabled:

1. Power on the system, and the primary core initializes hardware components and executes the bootloader (e.g., Trusted Firmware-A for Arm-based systems).
2. The bootloader loads and starts the operating system on the primary core.
3. The operating system initializes its kernel, essential components, and device drivers on the primary core.
4. After the primary core has completed its initialization, the operating system enables the secondary cores. This process is often referred to as "core bring-up" or "CPU hotplug."
5. The operating system assigns tasks or processes to the secondary cores, allowing them to participate in executing applications and system processes.

It is important to note that the specific process for enabling secondary cores depends on the hardware platform, firmware, bootloader, and operating system in use. Some systems may require additional configuration or software support to enable secondary cores.

Question: `[22:58]` Does TF-A support Arm32 or Arm64?

Answer:

Trusted Firmware-A (TF-A) supports both Arm32 (AArch32) and Arm64 (AArch64) architectures. It is designed to provide a reference implementation of Secure World software for Armv8-A and Armv7-A processors. TF-A aims to cater to various security and non-security use cases, and it is built to be adaptable and portable across different Arm architectures.

Question: `[22:59]` Which types of ARM cores is supported by TF-A?

Answer:

Trusted Firmware-A (TF-A) supports Armv8-A and Armv7-A processors. The Armv8-A architecture encompasses both the 64-bit (AArch64) and the 32-bit (AArch32) execution states, while the Armv7-A architecture is a 32-bit (AArch32) execution state. TF-A provides a reference implementation of Secure World software for a variety of Arm Cortex-A cores within these architectures, including Cortex-A53, Cortex-A57, Cortex-A72, Cortex-A75, and Cortex-A76, among others.

Question: `[23:01]` Does Armv7-A has 64 bits?

Answer:

No, Armv7-A architecture does not have 64 bits. Armv7-A is a 32-bit (AArch32) execution state architecture. Armv8-A is the architecture that introduces both 64-bit (AArch64) and 32-bit (AArch32) execution states.

Question: `[23:05]` Must AArch64 have TF-A?

Answer:

No, AArch64 does not require TF-A (Arm Trusted Firmware-A). TF-A is an open-source project that provides a reference implementation of the software components needed for the secure boot and runtime firmware on Armv8-A platforms. While it is commonly used and recommended for secure firmware development on Armv8-A platforms, it is not mandatory. You can use other firmware solutions or develop custom secure firmware for AArch64 platforms, depending on your requirements and platform constraints.

Question: `[23:13]` Does Armv8-A has 32 bits?

Answer:

Yes, Armv8-A architecture supports both 64-bit (AArch64) and 32-bit (AArch32) execution states. While the primary focus of Armv8-A is the introduction of the AArch64 execution state, it also includes enhancements to the AArch32 state, which is compatible with the Armv7-A architecture. This allows for flexibility and backwards compatibility in systems implementing Armv8-A architecture.

Question: `[23:23]` What is “Arm A-profile”?

Answer:

The Arm A-profile is a family of Arm CPU architectures that are designed for high-performance and power-efficient applications, including mobile devices, consumer electronics, servers, and networking infrastructure. The A-profile CPUs are based on the Armv7-A and Armv8-A architectures, which support both 32-bit (AArch32) and 64-bit (AArch64) execution states. The A-profile processors provide a wide range of performance and power options to suit various use cases and include popular cores like Cortex-A series (e.g., Cortex-A72, Cortex-A57, Cortex-A53, etc.).

Question: `[23:24]` Does “Arm Firmware Framework for Arm A-profile” document is only about Cortex-A?

Answer:

The “Arm Firmware Framework for Arm A-profile” document is focused on the Arm A-profile architecture, which includes Cortex-A series processors. However, it is not limited to Cortex-A processors only. The document provides guidelines and specifications for developing firmware for Arm A-profile processors in general, including Cortex-A series processors and potentially other A-profile processors developed by Arm or its partners. It aims to ensure a consistent approach to firmware development, security, and functionality across different Arm A-profile implementations.

Arm.0410.md

Question: `[23:15]` What are types of ARM CPU? Please also describe them in a table with 32 bits and 64 bits.

Answer:

ARM CPUs come in a variety of architectures and designs, each tailored to specific needs and applications. Some common types of ARM CPUs are Cortex-A, Cortex-R, and Cortex-M series. Here’s a table describing these three series along with their support for 32-bit and 64-bit execution states:

Series	Target Applications	Description	32-bit	64-bit
Cortex-A	High-performance systems	Designed for high-performance devices like smartphones, tablets, and other consumer electronics, as well as servers. These processors provide high performance and power efficiency, suitable for running complex operating systems like Linux, Android, and iOS.	Yes (Armv7-A)	Yes (Armv8-A)
Cortex-R	Real-time systems	Targeted towards real-time systems that require fast, deterministic response times like automotive electronics, industrial control systems, and medical devices. These processors prioritize reliability, fault tolerance, and real-time performance.	Yes (Armv7-R)	No*
Cortex-M	Low-power microcontrollers	Optimized for low-power, cost-sensitive applications like IoT devices, wearables, and other embedded systems. These processors are designed to offer low-latency response times, energy efficiency, and ease of use.	Yes (Armv6-M, Armv7-M, Armv8-M Baseline)	Yes (Armv8-M Mainline)

Please note that within each series, there are multiple specific processor models with different performance levels, features, and target applications. The examples provided above are not exhaustive.

*As of my knowledge cutoff in September 2021, Cortex-R series does not have 64-bit variants. However, it's essential to check ARM's official resources for any updates or new additions to the Cortex-R series.

Question: `[23:18]` Does Armv8-A has 32 bits?

Answer:

Yes, Armv8-A architecture includes both 64-bit (AArch64) and 32-bit (AArch32) execution states. The 32-bit execution state in Armv8-A is largely compatible with the Armv7-A architecture, which allows for backward compatibility with existing 32-bit software.

In AArch32 state, Armv8-A processors support the ARM and Thumb instruction sets, which are commonly used in previous 32-bit Arm architectures. This dual execution state design allows Armv8-A based systems to run a mix of 32-bit and 64-bit applications and operating systems, providing flexibility and a smooth transition for software ecosystems.

Question: `[23:19]` Does Armv7-A has 64 bits?

Answer:

No, Armv7-A architecture does not have 64-bit support. Armv7-A is a 32-bit architecture that was used in many smartphones, tablets, and other devices before the introduction of Armv8-A. Armv7-A supports the ARM and Thumb instruction sets and includes features like NEON SIMD (Single Instruction, Multiple Data) and hardware floating-point support.

Armv8-A is the successor to Armv7-A and introduced the 64-bit execution state (AArch64) alongside the 32-bit execution state (AArch32) for backward compatibility with Armv7-A software. The 64-bit support in Armv8-A allows for improved performance, larger addressable memory, and additional architectural enhancements compared to Armv7-A.
