2023-04-13

Created: 2023-04-26 23:54:14

Questions

- Types of ARM CPU `(230410.Arm.txt)`
  - `[09:13]` Below ariticle “In aarch32 state, what is the mechanism to switch to aarch64 in software?” come from `(Arm.0410.md)`
  - `[09:17]` How do I specify only AArch32 of Armv8-A at run-time? `(Arm.0410.md)`
  - `[09:34]` Is uboot 32-bit or 64-bit? `(Arm.0410.md)`
- 1 - Introduction `(230413.EME.txt)`
  - `[11:06]` The below text is “1. Introduction” chapter of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`
  - `[11:13]` What is CDN? `(EME.0413.md)`
  - `[11:15]` What is Media Stack? `(EME.0413.md)`
  - `[11:17]` What is MediaKeySession? `(EME.0413.md)`
  - `[11:19]` Which components does MediaKeySession contain? `(EME.0413.md)`
  - `[11:20]` Does Media Stack contain many MediaKeySession? `(EME.0413.md)`
  - `[11:22]` Where do frames exist in Encrypted Media Extensions? `(EME.0413.md)`
- Abstract & Status of This Document `(230413.EME.txt)`
  - `[11:07]` The below text is “Abstract” and “Status of This Document” chapters of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`
- 2 - Definitions `(230413.EME.txt)`
  - `[11:10]` The below text is “2. Definitions” chapter of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`
  - `[11:11]` The below text is the second part of “2. Definitions” chapter of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`
- Memory `(230327.OP-TEE.txt)`
  - `[17:07]` Who can allocate SMAF memory? TA or CA `(OP-TEE.0327.md)`
- Basic Concept `(230402.DRM.txt)`
  - `[17:15]` What is secure data path? `(DRM.0402.md)`
- Basic Concepts `(230327.OP-TEE.txt)`
  - `[18:01]` What is Op-TEE Dispatcher? `(OP-TEE.0327.md)`
- FF-A, TF-A, and TrustZone `(230409.FF-A.txt)`
  - `[21:09]` What is SMCCC? `(FF-A.0409.md)`
  - `[22:45]` What is SPL? `(FF-A.0409.md)`
  - `[22:56]` What is fip.bin? `(FF-A.0409.md)`
- Qualcomm `(230407.OP-TEE.History.txt)`
  - `[21:37]` What is SPU? `(OP-TEE.History.0407.md)`
  - `[21:40]` Does SPU have secret keys? `(OP-TEE.History.0407.md)`
- Development `(230327.OP-TEE.txt)`
  - `[21:54]` What is libteec.so? `(OP-TEE.0327.md)`
  - `[21:55]` Does libteec.so run in EL0 or EL1? `(OP-TEE.0327.md)`
  - `[21:56]` Does optee.ko run in EL1? `(OP-TEE.0327.md)`
  - `[21:58]` What is TEE supplicant? `(OP-TEE.0327.md)`
- Privilege Levels `(230329.TrustZone.txt)`
  - `[22:31]` Are EL3 and S-EL3 same? `(TrustZone.0329.md)`
- PKCS#11 `(230327.OP-TEE.txt)`
  - `[23:03]` What is HUK? `(OP-TEE.0327.md)`
  - `[23:04]` So what is TSK? `(OP-TEE.0327.md)`

Q & A

Types of ARM CPU `(230410.Arm.txt)`

Question: `[09:13]` Below ariticle “In aarch32 state, what is the mechanism to switch to aarch64 in software?” come from `(Arm.0410.md)`

Answer:

The discussion in the linked forum post provides a deeper insight into how execution states can be switched between AArch32 and AArch64 in ARMv8-A architecture.

To summarize the key points from the discussion:

1. Any exception type in AArch32 state could potentially lead to the execution state changing to AArch64. However, no exception taken while in AArch64 will result in the execution state changing to AArch32.
2. For exception returns, an exception return in AArch64 might cause the execution state to change to AArch32. No exception return in AArch32 can lead to the execution state changing to AArch64, except indirectly through triggering another exception that causes the switch.
3. A change of execution state can only occur if there is also a change in Exception Level (EL). For example, an exception from EL0 to EL1 could lead to a change in the execution state, but an exception from EL1 to EL1 could not.
4. The ARMv8-A architecture manual provides pseudocode for handling various exceptions and switching between AArch32 and AArch64 states. The manual can be referred to for understanding the specific conditions for each type of exception.

The exact mechanism for switching between AArch32 and AArch64 execution states depends on the exception type and the system configuration, and can be found in the ARMv8-A architecture manual.

Question: `[09:17]` How do I specify only AArch32 of Armv8-A at run-time? `(Arm.0410.md)`

Answer:

In ARMv8-A architecture, you can use the Execution state (AArch32 or AArch64) depending on the current Exception Level (EL) and the settings of specific configuration registers. Here’s how you can ensure the system operates in AArch32 mode at runtime:

1. Configure the relevant registers: - If you are in the Secure state, you need to set the `SCR_EL3.RW` bit to 0. - If you are in the Non-secure state, you need to set the `HCR_EL2.RW` bit to 0.
2. When executing in the context of an OS or hypervisor, ensure that your software configures the registers correctly to maintain the AArch32 mode. This might include configuring the `SCTLR_ELx` register and other system registers as needed.
3. When running applications, ensure they are compiled for AArch32 mode. When an exception occurs (e.g., an interrupt or a system call), and the system transitions to a higher EL, it will remain in AArch32 mode if the settings mentioned in step 1 are properly configured.

By following these steps, you ensure that your ARMv8-A system operates in AArch32 mode at runtime. Keep in mind that, depending on your use case, certain performance and features available in AArch64 mode might not be accessible when operating in AArch32 mode.

Question: `[09:34]` Is uboot 32-bit or 64-bit? `(Arm.0410.md)`

Answer:

U-Boot, or the Universal Bootloader, is a widely used open-source bootloader that can be built for various target platforms and architectures. It can be built for both 32-bit and 64-bit ARM architectures depending on the target system and its requirements.

When building U-Boot for a specific ARM platform, you can configure it to support either 32-bit (AArch32) or 64-bit (AArch64) execution states, depending on the underlying hardware and your requirements. If your target system is based on ARMv8-A or a newer architecture, you may choose to build U-Boot in 64-bit mode to take advantage of the features and performance improvements offered by the AArch64 execution state.

In summary, U-Boot can be built as either 32-bit or 64-bit, depending on the target platform and requirements. The specific execution state depends on the build configuration and the target hardware capabilities.

---

1 - Introduction `(230413.EME.txt)`

Question: `[11:06]` The below text is “1. Introduction” chapter of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`

Answer: … …

Question: `[11:13]` What is CDN? `(EME.0413.md)`

Answer:

A Content Delivery Network (CDN) is a system of distributed servers that deliver web content, such as images, videos, stylesheets, and scripts, to users based on their geographical location. The primary goal of a CDN is to improve the performance and availability of web content by reducing the latency experienced by users when they access resources on the internet.

When a user requests a resource, the CDN directs the request to the nearest server (also known as an edge server) in its network, minimizing the distance the data needs to travel and thus reducing latency. CDNs are commonly used by websites with a global audience to ensure faster loading times and a better user experience.

In addition to reducing latency, CDNs can also improve reliability and security by distributing the load across multiple servers and providing protection against DDoS attacks and other web threats.

Question: `[11:15]` What is Media Stack? `(EME.0413.md)`

Answer:

A Media Stack refers to the collection of software components, protocols, and technologies used to process, transmit, and display multimedia content, such as audio and video, on various devices. The media stack is responsible for handling tasks such as encoding, decoding, streaming, buffering, and rendering media content in a way that ensures a smooth and seamless user experience.

A typical media stack consists of several layers, including:

1. Application Layer: This layer contains the user interface and interacts with the user, enabling them to access and control the media content.
2. Media Framework Layer: This layer manages the playback of media content and provides APIs for application developers to build media-related features.
3. Codec Layer: This layer handles the encoding and decoding of media content. Codecs (encoder-decoder) are used to compress and decompress audio and video data to enable efficient transmission and storage.
4. Transport Layer: This layer is responsible for transmitting media content over a network. It includes protocols like Real-Time Messaging Protocol (RTMP), Dynamic Adaptive Streaming over HTTP (DASH), and HTTP Live Streaming (HLS) for streaming media content over the internet.
5. Operating System and Hardware Layer: This layer includes the operating system, device drivers, and hardware components that interact with the media stack. It ensures proper integration of media playback with the device’s hardware, such as audio and video output systems.

These layers work together to provide a seamless media playback experience for users, regardless of the device or network conditions. The media stack must be optimized for performance, scalability, and compatibility to handle various media formats, codecs, and streaming protocols.

Question: `[11:17]` What is MediaKeySession? `(EME.0413.md)`

Answer:

`MediaKeySession` is an interface in the Encrypted Media Extensions (EME) API, which is a part of the HTML5 standard. The EME API enables web applications to play encrypted media content by providing a means for key exchange and license management between the web application, the Content Decryption Module (CDM), and a license server.

A `MediaKeySession` object represents a single session between the web application and the CDM. Each session is associated with a unique set of keys or licenses needed to decrypt the protected media content. The `MediaKeySession` interface provides methods for generating and exchanging messages between the web application and the CDM, allowing the CDM to request and receive decryption keys from a license server.

The typical steps for using `MediaKeySession` are:

1. Create a `MediaKeys` object, which represents the CDM, and associate it with an HTML5 `HTMLMediaElement` (e.g., a `<video>` element).
2. Create a `MediaKeySession` object by calling the `createSession()` method on the `MediaKeys` object.
3. Generate a license request by calling the `generateRequest()` method on the `MediaKeySession` object with the required Initialization Data (obtained from the encrypted media content).
4. Send the license request to the license server.
5. The license server responds with the required decryption keys or licenses.
6. Update the `MediaKeySession` with the decryption keys or licenses received from the license server using the `update()` method.

Once the `MediaKeySession` has been updated with the appropriate decryption keys or licenses, the media content can be decrypted by the CDM and played back by the `HTMLMediaElement`. The `MediaKeySession` also provides methods for managing key expiration and handling session close events.

Question: `[11:19]` Which components does MediaKeySession contain? `(EME.0413.md)`

Answer:

The `MediaKeySession` interface in the Encrypted Media Extensions (EME) API does not “contain” components in a hierarchical sense, but it does provide several properties and methods that allow web applications to interact with Content Decryption Modules (CDMs) for handling encrypted media content. Here are the key properties and methods associated with `MediaKeySession`:

1. Properties: - `sessionId`: A read-only property that represents a unique identifier for the session. It is generated by the CDM and can be used to reference the session for subsequent operations.

2. Methods: - `generateRequest(initDataType, initData)`: Generates a license request based on the provided Initialization Data (`initData`) and its type (`initDataType`). This method returns a Promise that resolves once the request is created, allowing the application to send the request to the appropriate license server. - `load(sessionId)`: Loads data for a specified `sessionId` into a new session. This method is useful for loading previously persisted session data, like when resuming playback after a page refresh. It returns a Promise that resolves with a boolean indicating if the data was successfully loaded or not. - `update(response)`: Updates the session with the license or key data received from the license server. The `response` parameter contains the data. This method returns a Promise that resolves once the session has been updated with the new data. - `close()`: Closes the session and releases any resources associated with it, such as decryption keys. This method returns a Promise that resolves when the session is closed. - `remove()`: Removes any license or key data associated with the session, making the keys no longer usable for decryption. This method returns a Promise that resolves when the data has been removed.

3. Event Handlers: - `onkeystatuseschange`: An event handler that gets called when key status changes occur, such as when a key expires or is released. - `onmessage`: An event handler that gets called when the CDM sends a message to the web application, usually related to license requests or errors.

These properties, methods, and event handlers work together to enable web applications to manage encrypted media sessions, request licenses, update keys, and handle related events.

Question: `[11:20]` Does Media Stack contain many MediaKeySession? `(EME.0413.md)`

Answer:

The term “Media Stack” refers to the collection of components, technologies, and interfaces that are involved in media processing and playback in a web browser. This includes components like media codecs, streaming protocols, and APIs like HTMLMediaElement, Media Source Extensions (MSE), and Encrypted Media Extensions (EME).

In the context of EME and encrypted media playback, a Media Stack may interact with multiple MediaKeySessions. Each MediaKeySession represents a unique session for managing encrypted media keys, licenses, and associated events.

During the playback of encrypted media, an application can create and manage multiple MediaKeySessions for different purposes, such as handling different pieces of content, managing multiple keys or licenses, or dealing with different encryption schemes. So, in this sense, the Media Stack can indeed work with multiple MediaKeySessions.

However, it is essential to understand that the Media Stack does not “contain” MediaKeySessions in a hierarchical sense, but rather interacts with them as part of the overall media processing and playback workflow.

Question: `[11:22]` Where do frames exist in Encrypted Media Extensions? `(EME.0413.md)`

Answer:

In the context of Encrypted Media Extensions (EME), frames refer to the individual video or audio samples that are part of the media stream. EME does not directly handle frames but focuses on the encryption and decryption aspect of media content.

Frames exist within the encrypted media container, such as MPEG-DASH or HLS. These media containers have segments, which contain encrypted frames. The media container’s role is to package the video and audio frames along with the necessary metadata required for playback, including encryption-related information like initialization data.

When the media player requests a segment, it processes the encrypted frames and passes the encryption-related information to the EME. The EME then handles the necessary key management, license acquisition, and decryption of content using the Content Decryption Module (CDM). Once the frames are decrypted, they are passed back to the media player for decoding and rendering.

In summary, frames exist within the encrypted media container, and EME manages the process of decryption but does not directly handle the frames themselves. The media player, together with EME and the underlying Content Decryption Module (CDM), work cohesively to handle, decrypt, and render the frames for playback.

Abstract & Status of This Document `(230413.EME.txt)`

Question: `[11:07]` The below text is “Abstract” and “Status of This Document” chapters of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`

Answer: … …

---

2 - Definitions `(230413.EME.txt)`

Question: `[11:10]` The below text is “2. Definitions” chapter of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`

Answer: … …

Question: `[11:11]` The below text is the second part of “2. Definitions” chapter of https://www.w3.org/TR/encrypted-media `(EME.0413.md)`

Answer: … …

Memory `(230327.OP-TEE.txt)`

Question: `[17:07]` Who can allocate SMAF memory? TA or CA `(OP-TEE.0327.md)`

Answer:

In the context of OP-TEE (Open Portable Trusted Execution Environment), the SMAF (Secure Memory Allocation Framework) is used to allocate memory for TAs (Trusted Applications).

Typically, the allocation of SMAF memory is managed by the OP-TEE OS (Trusted OS) and is not directly accessible by the CAs (Client Applications) running in the normal world. The reason for this is to maintain a strict separation between the secure and normal worlds, which is essential for the security and isolation of the trusted applications.

When a CA wants to communicate with a TA, it does so using the OP-TEE client API, which allows the CA to send messages and data to the TA. The OP-TEE OS then handles the secure allocation of memory for the TA, ensuring that the sensitive data and code execution remain protected within the secure environment.

In summary, the SMAF memory allocation is typically managed by the OP-TEE OS and is intended for use by TAs, while CAs interact with TAs through the OP-TEE client API.

Basic Concept `(230402.DRM.txt)`

Question: `[17:15]` What is secure data path? `(DRM.0402.md)`

Answer:

A secure data path, in the context of digital content protection and DRM (Digital Rights Management) systems, refers to a protected and encrypted communication channel used for transmitting sensitive information, such as content keys, licenses, and encrypted digital content. The secure data path helps to ensure that the data remains confidential and tamper-proof as it moves through various stages of processing, from content delivery to rendering or playback on a user's device.

A secure data path typically involves several key components:

1. Encryption: Sensitive data is encrypted using strong cryptographic algorithms, ensuring that it remains confidential and unreadable to unauthorized parties during transmission and storage.
2. Authentication: Both the sender and receiver of the data must prove their identities using cryptographic techniques, such as digital signatures or certificates, to ensure that the data is only shared with authorized parties.
3. Secure communication protocols: Protocols like TLS (Transport Layer Security) or DTLS (Datagram Transport Layer Security) are used to establish secure communication channels between the sender and receiver, providing confidentiality, integrity, and authentication for the transmitted data.
4. Hardware-based security: In some cases, secure data paths may utilize hardware-based security features, such as Trusted Execution Environments (TEEs) or secure enclaves, to further protect sensitive data and cryptographic operations from potential software attacks or vulnerabilities.
5. End-to-end protection: A secure data path aims to maintain the confidentiality and integrity of the data throughout its entire lifecycle, from content creation and encryption to delivery, decryption, and playback on the end-user's device.

By implementing a secure data path, content providers and DRM systems can protect sensitive information and digital content from unauthorized access, tampering, or eavesdropping, ensuring that the content is only accessible and consumable by authorized users or devices according to the specified usage rights and restrictions.

Basic Concepts `(230327.OP-TEE.txt)`

Question: `[18:01]` What is Op-TEE Dispatcher? `(OP-TEE.0327.md)`

Answer:

The OP-TEE dispatcher, sometimes referred to as the "Secure Monitor Call (SMC) dispatcher" or simply "dispatcher," is a

component within the OP-TEE Trusted Execution Environment that handles communication between the normal world (non-secure) and the secure world (OP-TEE environment).

When a client application (CA) in the normal world wants to communicate with a trusted application (TA) in the secure world, it sends a request through the OP-TEE driver (opteed) in the form of an SMC call. The SMC call is then intercepted by the OP-TEE dispatcher, which is responsible for processing the call, routing it to the appropriate TA, and returning the results back to the caller.

The dispatcher is responsible for managing the communication and ensuring the security and isolation of the TAs from the normal world, as well as handling secure world tasks such as opening and closing sessions, invoking commands, and managing shared memory.

In summary, the OP-TEE dispatcher is a crucial component that bridges the gap between the normal world and the secure world, ensuring secure communication and execution of trusted applications within the OP-TEE environment.

FF-A, TF-A, and TrustZone `(230409.FF-A.txt)`

Question: `[21:09]` What is SMCCC? `(FF-A.0409.md)`

Answer:

The SMCCC (Secure Monitor Call Calling Convention) is a standardized calling convention for ARM-based systems, which defines how Secure Monitor Calls (SMCs) are used to request services from a secure monitor, such as Trusted Firmware-A (TF-A). SMCCC provides a consistent interface for communication between the non-secure world (e.g., an operating system) and the secure world (e.g., a trusted execution environment or firmware). This standardized interface allows for better interoperability between different software components and platforms in ARM-based systems.

Question: `[22:45]` What is SPL? `(FF-A.0409.md)`

Answer:

SPL (Secondary Program Loader) is a term used in the context of embedded systems and bootloaders. It is a small, lightweight bootloader that runs before the primary bootloader and is responsible for initializing essential hardware components, such as clocks, memory, and peripherals. After performing the required initialization, the SPL loads the primary bootloader (e.g., U-Boot) into memory and transfers control to it, allowing the primary bootloader to continue the boot process and eventually load the operating system. The use of an SPL can help improve boot time, reduce the complexity of the primary bootloader, and provide a uniform interface for different hardware platforms.

Question: `[22:56]` What is fip.bin? `(FF-A.0409.md)`

Answer:

`fip.bin` (Firmware Image Package) is a binary file used in Arm Trusted Firmware-A (TF-A) for packing multiple firmware images into a single file. It is used as a part of the boot process in Arm-based systems, particularly in the context of Armv8-A architecture.

The `fip.bin` file typically contains a collection of firmware images, such as:

1. BL1 (First Level Bootloader): Responsible for initial platform setup and loading BL2.
2. BL2 (Second Level Bootloader): Responsible for loading BL3x images.
3. BL3-1 (EL3 Runtime Firmware): Provides runtime services, such as PSCI (Power State Coordination Interface), to the lower ELs (Exception Levels).
4. BL3-2 (Trusted OS Firmware): The Secure-EL1 Payload, typically the TrustZone-based OS or a secure monitor.
5. BL3-3 (Non-Trusted Firmware): The Non-secure-EL1 Payload, usually the main OS bootloader like U-Boot or GRUB.

The `fip.bin` file is created using the `fiptool` utility provided by Arm Trusted Firmware. The tool combines the various firmware images into a single binary, which can then be loaded and executed by the platform's bootloader during the boot process. This approach simplifies firmware management and distribution while also enabling the bootloader to locate and authenticate the required images more easily.

Qualcomm `(230407.OP-TEE.History.txt)`

Question: `[21:37]` What is SPU? `(OP-TEE.History.0407.md)`

Answer:

The Secure Processing Unit (SPU) is a hardware-based security component developed by Qualcomm for their Snapdragon SoCs (System on a Chip). Introduced in 2017, the SPU functions as a Trusted Execution Environment (TEE) and leverages ARM TrustZone technology to create an isolated secure environment within the processor. This secure environment is designed to protect sensitive data, execute security-related tasks, and provide an additional layer of hardware-based security.

The SPU is a separate subsystem within the Snapdragon SoCs, distinct from the main CPU. By being a dedicated hardware component, it offers enhanced security features, ensuring that sensitive data and operations are isolated from potential vulnerabilities in the main CPU or other system components. This helps prevent unauthorized access or tampering, even if other parts of the system are compromised.

Question: `[21:40]` Does SPU have secret keys? `(OP-TEE.History.0407.md)`

Answer:

Yes, the Secure Processing Unit (SPU) in Qualcomm's Snapdragon SoCs can store and manage secret keys. The SPU is designed to handle sensitive data, including cryptographic keys, ensuring that they are protected from unauthorized access or tampering.

The SPU provides hardware-based security features, such as secure key storage, secure boot, and cryptographic operations, all within an isolated environment separate from the main CPU. By securely managing secret keys and performing encryption and decryption operations within the SPU, it helps maintain the confidentiality and integrity of sensitive data, even if other parts of the system are compromised.

In summary, the SPU in Qualcomm's Snapdragon SoCs is designed to store and manage secret keys securely, providing an additional layer of hardware-based security for sensitive data and operations.

Development `(230327.OP-TEE.txt)`

Question: `[21:54]` What is libteec.so? `(OP-TEE.0327.md)`

Answer:

`libteec.so` is a shared library in the OP-TEE (Open Portable Trusted Execution Environment) system, which is part of the client-side implementation of the GlobalPlatform TEE Client API. This library allows client applications (CAs) in the normal world to communicate with trusted applications (TAs) running in the secure world, i.e., inside the TEE.

The TEE Client API specifies a set of functions that client applications can use to open and close sessions with trusted applications, to send commands and data to the TAs, and to manage shared memory between the normal world and the secure world. `libteec.so` provides the implementation of these functions, handling the low-level communication with the OP-TEE OS, usually through a dedicated driver.

In summary, `libteec.so` is a crucial component that enables applications running in the normal world to utilize the secure services provided by trusted applications within the OP-TEE environment.

Question: `[21:55]` Does libteec.so run in EL0 or EL1? `(OP-TEE.0327.md)`

Answer:

`libteec.so` runs in EL0 (Exception Level 0) since it is a user-space library that is part of client applications (CAs) in the normal world. EL0 is the lowest privilege level and is where applications run on ARM-based systems.

EL1 (Exception Level 1) is the level at which operating system kernel code runs, and it has a higher privilege level than EL0. The OP-TEE OS, which manages trusted applications (TAs) in the secure world, runs at a higher exception level (EL1 in

AArch64 or S-EL1 in AArch32) to ensure isolation between the normal and secure worlds. However, `libteec.so` is not part of the OP-TEE OS; it is a user-space library for communicating with the TEE from the normal world.

Question: `[21:56]` Does optee.ko run in EL1? `(OP-TEE.0327.md)`

Answer:

`optee.ko` is a Linux kernel module that runs in the normal world. Since it is part of the kernel, it runs at EL1 (Exception Level 1) on AArch64 systems. EL1 is the level at which operating system kernel code runs, and it has a higher privilege level than EL0, where user-space applications run.

However, it is important to note that `optee.ko` runs in the normal world, not in the secure world. It serves as a driver to facilitate communication between client applications (CAs) in the normal world and the OP-TEE OS in the secure world. The actual OP-TEE OS runs at a higher exception level (EL1 in AArch64 or S-EL1 in AArch32) in the secure world to ensure isolation between the normal and secure worlds.

Question: `[21:58]` What is TEE supplicant? `(OP-TEE.0327.md)`

Answer:

The TEE (Trusted Execution Environment) supplicant is a user-space daemon in the normal world that works alongside the OP-TEE environment. It serves as an intermediary between the TEE OS and the normal world's user-space applications, facilitating communication and providing additional services for TEE operations.

Some of the primary responsibilities of the TEE supplicant include:

1. Loading and managing shared libraries for Trusted Applications (TAs).
2. Handling communication with storage services, such as file access and data persistence, for TAs.
3. Assisting with inter-process communication (IPC) between client applications (CAs) and TAs.
4. Managing cryptographic operations that require access to the normal world.

The TEE supplicant is an essential component of the OP-TEE ecosystem, working in tandem with the kernel-side driver (e.g., optee.ko) and user-space TEE client libraries (e.g., libteec.so) to enable a seamless, secure execution environment for sensitive applications.

Privilege Levels `(230329.TrustZone.txt)`

Question: `[22:31]` Are EL3 and S-EL3 same? `(TrustZone.0329.md)`

Answer:

No, EL3 and S-EL3 are not the same. EL3 refers to the Exception Level 3, which is the highest privilege level in the ARMv8-A architecture. It is typically used by the Secure Monitor or firmware running in the Secure World.

S-EL3, on the other hand, is an abbreviation for Secure Exception Level 3. It refers to EL3 when the processor is operating in the Secure World. The Secure World has its own set of Exception Levels (S-EL0, S-EL1, and S-EL3), which operate independently of the Non-secure World's Exception Levels (EL0, EL1, and EL2).

While both EL3 and S-EL3 are related to the Secure World, the difference lies in their representation:

- EL3: Refers to Exception Level 3 in general, without explicitly specifying the security state.
- S-EL3: Explicitly refers to Exception Level 3 in the Secure World.

PKCS#11 `(230327.OP-TEE.txt)`

Question: `[23:03]` What is HUK? `(OP-TEE.0327.md)`

Answer:

HUK stands for Hardware Unique Key. It is a unique, device-specific key that is generated and stored within a device's hardware during manufacturing. HUKs are used to provide a root of trust and ensure secure device identification and authentication. In the context of OP-TEE, the Hardware Unique Key can be used to derive other cryptographic keys to secure various aspects of the system, such as secure storage or secure communication.

Question: `[23:04]` So what is TSK? `(OP-TEE.0327.md)`

Answer:

TSK stands for Trusted Storage Key. In the context of OP-TEE, it is a key used to encrypt and authenticate the data stored in the secure storage. The TSK is derived from the Hardware Unique Key (HUK) and other device-specific information, which ensures that the encrypted data can only be accessed and decrypted by the same device. This provides a secure way to store sensitive data, such as cryptographic keys or user credentials, on the device.
