# 2023-03-29

Created: 2023-05-01 22:17:45

# Questions

- Normal World & Secure World `(230329.TrustZone.yaml)`
  - `[08:51]` Can Normal world and Secure World run at the same time in ARM? `(TrustZone.0329.md)`
  - `[08:52]` But can Normal world and Secure World run at the same time in multi processors of ARM? `(TrustZone.0329.md)`
- Memory `(230329.TrustZone.yaml)`
  - `[08:53]` How to define memory size of secure world? `(TrustZone.0329.md)`
  - `[08:53]` Can memory size of secure world be changed? `(TrustZone.0329.md)`
- SM & SPM & SPD `(230329.TrustZone.yaml)`
  - `[08:55]` What is SMC? `(TrustZone.0329.md)`
  - `[08:57]` Can secure world request a service in normal world via SMC? `(TrustZone.0329.md)`
  - `[09:00]` What is SPD? `(TrustZone.0329.md)`
  - `[09:03]` Can SPD handle the request from the secure world? `(TrustZone.0329.md)`
  - `[09:05]` What is SPM? `(TrustZone.0329.md)`
  - `[09:41]` Why are SPD and SPM mutually exclusive? `(TrustZone.0329.md)`
- Secure Partition `(230329.TrustZone.yaml)`
  - `[09:04]` What is SP? `(TrustZone.0329.md)`
  - `[09:05]` Can different SPs communicate together? `(TrustZone.0329.md)`
  - `[09:06]` How to create a SP and when? `(TrustZone.0329.md)`
  - `[09:08]` Can I create a SP in run-time? `(TrustZone.0329.md)`
  - `[09:08]` What is TA? `(TrustZone.0329.md)`
  - `[09:24]` How do TAs and SPs communicate each other? `(TrustZone.0329.md)`
  - `[09:43]` Can a SP host many TAs? `(TrustZone.0329.md)`
  - `[09:44]` Can a TA communicate different SP? `(TrustZone.0329.md)`
- Privilege Levels `(230329.TrustZone.yaml)`
  - `[09:44]` What are EL0, EL1, EL2, S-EL0, S-EL1, S-EL2, S-EL3? `(TrustZone.0329.md)`
- Basic Concepts `(230328.TEE.yaml)`
  - `[10:04]` https://en.wikipedia.org/wiki/Trusted_execution_environment `(TEE.0328.md)`
  - `[13:55]` Please summarize Intel SGX and ARM TrustZone in tables. `(TEE.0328.md)`
  - `[13:58]` What are different between ARM SMC and Intel SMI? Please also summarize in a table. `(TEE.0328.md)`
  - `[13:59]` Please describe TEE in Intel. `(TEE.0328.md)`
  - `[14:00]` What are APIs of TEE? Please classify them. `(TEE.0328.md)`
  - `[14:02]` Can I add a myself TEE API? `(TEE.0328.md)`
- Memory `(230327.OP-TEE.yaml)`
  - `[16:19]` How does OP-TEE manage memory? `(OP-TEE.0327.md)`
  - `[16:27]` Are data in memory in secure world encrypted? `(OP-TEE.0327.md)`
- Signature `(230327.OP-TEE.yaml)`
  - `[16:44]` Is signature required when a normal world application request a service in secure world? `(OP-TEE.0327.md)`
- CA & TA `(230328.TEE.yaml)`
  - `[16:56]` How does an application in normal world request a service with shared memory of a trusted application in a secure world? Please describe the process with APIs. `(TEE.0328.md)`
  - `[17:10]` Which functions are used by TA (trusted application) in the above process? `(TEE.0328.md)`
  - `[17:16]` Could you please draw a sequence diagram of TEEC_InitializeContext, TEEC_OpenSession, TEEC_RegisterSharedMemory, TEEC_InvokeCommand, TEEC_ReleaseSharedMemory, TEEC_CloseSession, TEEC_FinalizeContext, TA_CreateEntryPoint, TA_DestroyEntryPoint, TA_OpenSessionEntryPoint,

TA_CloseSessionEntryPoint, TA_InvokeCommandEntryPoint `(TEE.0328.md)`

- `[17:25]` Is the above sequence described in GlobalPlatform? `(TEE.0328.md)`
- `[17:27]` Which document describe it? `(TEE.0328.md)`
- `[17:36]` Please give me two sample codes of the above sequence. One is for CA and another is for TA. `(TEE.0328.md)`
- `[17:40]` Please write sample code of TA again. The code should be complete. `(TEE.0328.md)`

---

# Q & A

## Normal World & Secure World `(230329.TrustZone.yaml)`

Question: `[08:51]` Can Normal world and Secure World run at the same time in ARM? `(TrustZone.0329.md)`

Answer:

In ARM TrustZone, the Normal world and Secure world cannot run concurrently. The processor can only execute code in either the Normal world or the Secure world at any given time. The two worlds are isolated from each other, and the processor switches between them based on events such as interrupts or requests for secure services. TrustZone uses a secure monitor mode to facilitate the transition between the two worlds, ensuring that the secure and non-secure states remain separate and protected.

---

Question: `[08:52]` But can Normal world and Secure World run at the same time in multi processors of ARM? `(TrustZone.0329.md)`

Answer:

Yes, in a multi-processor ARM system, it is possible for the Normal world and Secure world to run concurrently on separate cores. Each core in the system can be executing in either the Normal world or the Secure world independently. This allows for parallel processing of tasks in both worlds. However, it is important to note that the isolation between the two worlds is still maintained, and each core must follow the same TrustZone mechanisms to transition between the worlds when required.

---

## Memory `(230329.TrustZone.yaml)`

Question: `[08:53]` How to define memory size of secure world? `(TrustZone.0329.md)`

Answer:

Defining the memory size for the secure world in an ARM TrustZone-enabled system is highly dependent on the specific System-on-Chip (SoC) being used. Each SoC has different characteristics and mechanisms for allocating and partitioning memory resources between the secure and non-secure worlds.

Typically, the memory size for the secure world is defined during the system's initialization or configuration phase. The process involves configuring TrustZone-specific hardware components such as TrustZone Protection Controller (TZPC) and TrustZone Address Space Controller (TZASC) to set access permissions and memory partitions.

To define the memory size of the secure world, follow these general steps:

1. Determine the available memory resources: Consult your SoC's technical reference manual or datasheet to identify the available memory resources and their respective addresses.
2. Configure TZPC and TZASC: Based on the desired memory size for the secure world, configure the TZPC and TZASC registers to partition the memory accordingly. This may involve setting up specific regions and access permissions for secure and non-secure worlds. Some SoCs might provide alternative or additional mechanisms for memory partitioning.
3. Verify and update your secure world software: Ensure your secure world software is aware of the allocated memory size and adapts its memory management accordingly.

Please note that these steps are generalized, and you should refer to the documentation of your specific SoC for exact

procedures and register configurations.

---

Question: `[08:53]` Can memory size of secure world be changed? `(TrustZone.0329.md)`

Answer:

In most ARM TrustZone-enabled systems, the memory size of the secure world is typically defined during the system's initialization or configuration phase. However, whether the memory size of the secure world can be changed at runtime depends on the specific System-on-Chip (SoC) being used and its support for dynamic memory partitioning.

Some SoCs may allow for dynamic reconfiguration of TrustZone components like the TrustZone Protection Controller (TZPC) and TrustZone Address Space Controller (TZASC) during runtime, enabling changes to memory partitioning between the secure and non-secure worlds. In such cases, the secure world software would also need to support and be aware of such changes to adapt its memory management accordingly.

To determine if your specific SoC supports changing the memory size of the secure world at runtime, consult the technical reference manual or datasheet provided by the manufacturer. It's important to consider the potential security implications and any restrictions when dynamically changing memory partitioning, as it could introduce new attack surfaces or vulnerabilities.

---

## SM & SPM & SPD `(230329.TrustZone.yaml)`

Question: `[08:55]` What is SMC? `(TrustZone.0329.md)`

Answer:

SMC stands for Secure Monitor Call, which is an ARM instruction used in TrustZone-enabled systems. The SMC instruction is used to transfer control between the non-secure (Normal) world and the secure world. When the non-secure world needs to access a secure resource or service, it uses the SMC instruction to trigger a switch to the secure world. The secure world then handles the request, performs the necessary operations, and returns control back to the non-secure world using the ERET (Exception Return) instruction.

The SMC instruction ensures that transitions between the two worlds are securely managed and controlled, allowing the secure world to maintain its isolation and integrity. Secure Monitor Calls are essential for implementing security services and trusted applications that require isolation from the non-secure world while still providing a controlled interface for communication between the two worlds.

---

Question: `[08:57]` Can secure world request a service in normal world via SMC? `(TrustZone.0329.md)`

Answer:

Yes, the secure world can request a service in the normal world via SMC. Although it's more common for the normal world to request services from the secure world, the secure world may still need to access certain resources or services in the normal world. In such cases, the secure world can use SMC to make a request to the normal world.

However, it's important to note that the secure world should carefully manage any interactions with the normal world to avoid compromising its own security. When the secure world needs to access services or resources in the normal world, it should do so with the understanding that the normal world is potentially untrusted and might attempt to exploit any vulnerabilities in the communication process. Therefore, the implementation of such communication should be done cautiously, with a focus on maintaining the isolation and integrity of the secure world.

---

Question: `[09:00]` What is SPD? `(TrustZone.0329.md)`

Answer:

In the context of ARM TrustZone, SPD (Secure Payload Dispatcher) is a component responsible for managing and dispatching Secure Partition (SP) requests. An SPD serves as an intermediary layer between the normal world and secure world, allowing for the secure processing of service requests from the normal world.

When a normal world application requires services provided by the secure world, it issues a request via SMC (Secure Monitor

Call). The SPD receives this request, identifies the appropriate secure partition to handle the request, and dispatches the service execution to the corresponding partition. After the secure partition has processed the request, the SPD returns the result to the normal world.

By using an SPD, the secure world can maintain better isolation and security, as the SPD ensures that the normal world does not have direct access to secure partitions or their resources.

---

Question: `[09:03]` Can SPD handle the request from the secure world? `(TrustZone.0329.md)`

Answer:

Yes, the Secure Payload Dispatcher (SPD) can handle requests from both the normal world and the secure world. Although it is more common for the normal world to make service requests to the secure world, there might be cases where the secure world needs to access resources or services available in the normal world.

In such scenarios, the SPD plays a similar role in managing and dispatching these requests. It serves as an intermediary between the two worlds, ensuring that the communication is secure and that the appropriate services are invoked to handle the requests. This way, the secure world can access resources in the normal world without compromising its security and isolation.

---

Question: `[09:05]` What is SPM? `(TrustZone.0329.md)`

Answer:

SPM stands for Secure Partition Manager. In the context of ARM TrustZone or systems with secure partitions, the SPM is a component responsible for managing and orchestrating the execution of Secure Partitions (SPs). The SPM's role includes controlling access to resources, managing communication between SPs, ensuring the isolation of different secure partitions, and enforcing security policies.

In a nutshell, the Secure Partition Manager provides a framework for the secure world to manage the lifecycle and interactions of multiple secure partitions, ensuring that the security and isolation properties of each partition are maintained.

---

Question: `[09:41]` Why are SPD and SPM mutually exclusive? `(TrustZone.0329.md)`

Answer:

The Secure Partition Manager (SPM) and the Secure Partition Dispatcher (SPD) are two different approaches to managing secure partitions in the ARMv8.4-A architecture, and they are mutually exclusive because they serve different purposes and are intended for different use cases.

1. Secure Partition Manager (SPM): SPM is designed for the ARMv8.4-A architecture and is responsible for managing Secure Partitions (SPs). SPs are isolated execution environments running inside the secure world, which can host multiple security services or Trusted Applications (TAs). The SPM is responsible for creating, initializing, scheduling, and managing communication between SPs. It enables fine-grained partitioning of resources and services within the secure world.

2. Secure Partition Dispatcher (SPD): SPD is a part of ARM Trusted Firmware (ATF) and is used to manage secure services in ARMv8-A architecture. SPD provides an interface between the Normal World (NW) and the Secure World (SW). It is responsible for dispatching secure service requests from the NW to the appropriate secure execution environment, such as an SP or a TA, and returning the results back to the NW.

The reason they are mutually exclusive is that they cater to different architectural versions and use cases. SPM is designed specifically for ARMv8.4-A and provides advanced features for managing SPs, while SPD is used in ARMv8-A to manage secure services.

Using both SPM and SPD simultaneously would lead to conflicts in terms of resource allocation, scheduling, and communication between the secure world and the normal world. To avoid confusion and ensure proper secure partition management, either SPM or SPD is used, depending on the target ARM architecture and the specific use case.

---

## Secure Partition `(230329.TrustZone.yaml)`

Question: `[09:04]` What is SP? `(TrustZone.0329.md)`

Answer:

SP stands for Secure Partition, which is a software component or execution environment running within the secure world of ARM TrustZone. Secure Partitions are designed to provide isolation and protection for sensitive code and data, such as cryptographic operations, key management, or secure storage. The secure world hosts these SPs, which can communicate with the normal world through a well-defined interface, typically via SMC (Secure Monitor Call) instructions.

SPs help to ensure that critical operations and information remain protected from potentially malicious or insecure code running in the normal world. By isolating these components, TrustZone enhances the overall security of a system.

---

Question: `[09:05]` Can different SPs communicate together? `(TrustZone.0329.md)`

Answer:

Yes, different Secure Partitions (SPs) can communicate with each other within the secure world. However, this communication should be carefully designed and managed to maintain the isolation and security properties of each SP. Typically, an intermediary component like a secure partition manager or the secure monitor itself is responsible for facilitating communication between different SPs, ensuring that information sharing adheres to the security policies and requirements of each partition.

To enable secure communication between different SPs, developers should implement well-defined interfaces and access control policies to prevent unauthorized access or information leaks between partitions, maintaining the overall security goals of the system.

---

Question: `[09:06]` How to create a SP and when? `(TrustZone.0329.md)`

Answer:

Creating a Secure Partition (SP) typically involves a few steps and is usually done during the system initialization or when a secure application needs to be launched. Here are the steps to create an SP:

1. Design: Begin by identifying the secure functionality or application you want to isolate in a separate SP. Determine the security requirements, resource access, and communication interfaces needed for the secure partition.
2. Implementation: Implement the secure functionality or application, adhering to the security requirements and following best practices for secure programming. This may involve writing code and creating any necessary data structures, as well as implementing communication interfaces to interact with other SPs or the normal world.
3. Configuration: Configure the system to support the SP. This might involve configuring memory regions, peripherals, and access controls to meet the security requirements of the SP. This is typically done in the system's bootloader, firmware, or Secure Partition Manager (SPM), depending on the platform.
4. Initialization: Load and initialize the SP during system startup or when required. This process includes loading the secure partition's code and data into the protected memory regions and initializing the partition's state. The SPM or another trusted component may handle this initialization process.
5. Execution: Once initialized, the SP can be executed, providing its secure functionality to other components or applications in the system. The SPM or a trusted scheduler will manage the execution of the SP, ensuring proper isolation and adherence to security policies.

Creating an SP depends on the specific platform or system you are working with, as well as the requirements of your secure application. Ensure that you consult the relevant documentation and guidelines for your platform to create and manage secure partitions effectively.

---

Question: `[09:08]` Can I create a SP in run-time? `(TrustZone.0329.md)`

Answer:

Creating a Secure Partition (SP) during runtime is generally not recommended, as it could introduce security risks and may not be supported by some platforms. Secure Partitions are typically initialized during system startup or secure application launch to ensure proper isolation and security.

However, if your platform supports creating an SP during runtime and you have a valid use case, it might be possible. You would need to ensure that you take the necessary precautions to maintain the security of the system and the SPs, including:

1. Ensuring proper isolation between the newly created SP and the rest of the system, including other SPs.
2. Configuring memory regions, peripherals, and access controls correctly.
3. Verifying the integrity and authenticity of the code and data being loaded into the new SP.

Keep in mind that creating an SP during runtime may be more complex and error-prone than creating one during system startup or secure application launch. You should carefully consider the security implications and consult your platform's documentation to determine if this approach is supported and appropriate for your use case.

---

Question: `[09:08]` What is TA? `(TrustZone.0329.md)`

Answer:

TA stands for Trusted Application. In the context of ARM TrustZone or Trusted Execution Environments (TEEs), a Trusted Application is a secure piece of software running in the secure world. These applications are designed to handle sensitive data or perform security-critical operations, isolated from the normal world or non-secure applications.

Trusted Applications typically run on top of a TEE Operating System (TEE OS), which provides services like secure storage, cryptographic operations, and secure communication with the normal world. The TEE OS is responsible for managing and isolating multiple Trusted Applications, ensuring that they don't interfere with each other or leak sensitive data. Examples of TEE OSs include OP-TEE and Trustonic Kinibi.

Trusted Applications are usually loaded and instantiated by the TEE OS based on requests from the normal world or other secure world components. They can be used to perform a wide range of tasks, such as digital rights management, secure payment processing, biometric authentication, and more.

---

Question: `[09:24]` How do TAs and SPs communicate each other? `(TrustZone.0329.md)`

Answer:

TAs (Trusted Applications) and SPs (Secure Partitions) can communicate with each other using Inter-Partition Communication (IPC) mechanisms, which are platform-specific and depend on the underlying TEE (Trusted Execution Environment) OS or ARMv8.4-A architecture implementation.

When TAs and SPs need to communicate, they typically follow these steps:

1. Establish a communication channel: Both TAs and SPs should have a well-defined communication interface, such as shared memory or message passing. The implementation details depend on the TEE OS for TAs and the platform's support for SPs.
2. Secure the communication channel: To ensure the confidentiality and integrity of data exchanged between TAs and SPs, use cryptographic techniques, such as encryption and authentication.
3. Exchange messages: TAs and SPs can exchange messages through the established communication channel. These messages may include requests for services, data to be processed, or responses.
4. Synchronize communication: Depending on the IPC mechanism used, it may be necessary to implement synchronization primitives, such as semaphores or mutexes, to coordinate communication between TAs and SPs.
5. Handle errors and exceptions: Implement error handling and exception mechanisms to manage communication failures, timeouts, or other unexpected events.

It is essential to note that the communication between TAs and SPs requires careful implementation to ensure security and prevent potential attacks. Therefore, following best practices and guidelines for secure IPC is critical when developing

communication between TAs and SPs.

---

Question: `[09:43]` Can a SP host many TAs? `(TrustZone.0329.md)`

Answer:

A Secure Partition (SP) can host multiple Trusted Applications (TAs) in a single isolated execution environment within the secure world. The Secure Partition Manager (SPM) is responsible for managing and scheduling these TAs. Each TA can provide a specific security service, and they can all run within the same SP. By hosting multiple TAs within an SP, you can share resources among TAs, such as memory or peripherals, which can improve overall efficiency and reduce the overhead of managing individual TAs in separate SPs.

---

Question: `[09:44]` Can a TA communicate different SP? `(TrustZone.0329.md)`

Answer:

A Trusted Application (TA) can communicate with different Secure Partitions (SPs) through a secure communication channel, which is typically established by the Secure Partition Manager (SPM) or another trusted component in the secure world. The communication between TAs and different SPs is usually facilitated by a well-defined interface using messages or function calls. This allows TAs to request services or share data with other SPs in a controlled and secure manner, without directly accessing each other's memory or resources. However, it is important to note that the level of isolation between TAs and SPs may vary depending on the specific implementation and platform.

---

## Privilege Levels `(230329.TrustZone.yaml)`

Question: `[09:44]` What are EL0, EL1, EL2, S-EL0, S-EL1, S-EL2, S-EL3? `(TrustZone.0329.md)`

Answer:

In the ARMv8-A architecture, the privilege levels (EL) define different modes of operation for the processor, with each level having different access rights and capabilities. There are four Exception Levels (EL0, EL1, EL2, and EL3), and each level can operate in both Secure and Non-Secure states. Here's a brief explanation of each:

1. EL0: This is the least privileged level, where user-space applications run. It has limited access to system resources and cannot execute privileged instructions.
2. EL1: This level is where the operating system kernel runs. It has more privileges than EL0, enabling it to manage system resources, configure the hardware, and isolate user-space applications from each other.
3. EL2: This level is intended for hypervisors, which manage multiple virtual machines or operating systems. It provides the necessary isolation and control to manage virtualization.
4. EL3: This is the most privileged level, typically used for low-level firmware, such as a bootloader or a Trusted Execution Environment (TEE). It is responsible for setting up the system and ensuring a secure boot process.

The "S-" prefix indicates that the exception level is operating in the Secure state, which is part of ARM's TrustZone technology. TrustZone creates a separate, isolated environment for secure applications and data, ensuring that sensitive information remains protected from potential attacks or unauthorized access:

1. S-EL0: Secure User mode, where secure user-space applications run.
2. S-EL1: Secure Kernel mode, where the secure operating system kernel runs.
3. S-EL2: Secure Hypervisor mode, where a secure hypervisor can manage secure virtual machines.
4. S-EL3: Secure Monitor mode, where low-level firmware like a secure bootloader or a TEE operates.

In summary, the ARMv8-A architecture organizes the privilege levels (EL0, EL1, EL2, and EL3) and the Secure (S-EL0, S-EL1, S-EL2, and S-EL3) and Non-Secure states to create a robust and flexible environment for running a mix of secure and non-secure applications, operating systems, and hypervisors.

---

## Basic Concepts `(230328.TEE.yaml)`

Question: `[10:04]` https://en.wikipedia.org/wiki/Trusted_execution_environment `(TEE.0328.md)`

Answer:

A Trusted Execution Environment (TEE) is a secure area within the main processor of a device that ensures the confidentiality and integrity of code and data loaded within it. TEEs are designed to provide a higher level of security compared to traditional software execution environments by protecting sensitive information and functionality from external software attacks, hardware attacks, and physical tampering.

The main concept behind a TEE is to provide an isolated environment for executing security-sensitive tasks that cannot be accessed by other processes running on the same device. This isolation is achieved using hardware and software mechanisms that create a secure boundary around the TEE.

Some of the key features of TEEs include:

1. Isolation: TEEs provide a separate execution environment that isolates sensitive operations from the rest of the system.
2. Secure storage: TEEs provide a secure storage area for sensitive data, keys, and credentials that can only be accessed by the trusted applications running within the TEE.
3. Integrity: TEEs ensure the integrity of the code and data being executed by verifying the digital signatures of the applications before they are allowed to run within the TEE.
4. Confidentiality: TEEs protect the confidentiality of the data being processed within the environment by encrypting it.

TEEs have various applications, such as secure boot, digital rights management (DRM), and mobile payments. Some examples of TEE implementations include ARM TrustZone, Intel Software Guard Extensions (SGX), and AMD Secure Encrypted Virtualization (SEV).

It is important to note that while TEEs offer a higher level of security, they are not immune to attacks. Researchers have discovered vulnerabilities in various TEE implementations, emphasizing the need for ongoing research and development to enhance the security of these environments.

---

Question: `[13:55]` Please summarize Intel SGX and ARM TrustZone in tables. `(TEE.0328.md)`

Answer:

Here's a summary of Intel SGX and ARM TrustZone in table format:

| Feature | Intel SGX | ARM TrustZone |
|---|---|---|
| Full Name | Intel Software Guard Extensions | ARM TrustZone |
| Type | Hardware-based security feature | Hardware-based security feature |
| Purpose | Create secure enclaves for running trusted code & data | Create a secure environment for running trusted code & data |
| Architecture | x86-based processors | ARM-based processors |
| Isolation Method | Enclaves | Two worlds (Secure and Non-secure) |
| Memory Protection | Hardware-based encryption of enclave memory | Secure memory is isolated from the non-secure world |
| Remote Attestation | Supported, enables remote verification of enclave integrity | Supported, enables remote verification of secure world integrity |
| Applications | Secure computation, data protection, key management, etc. | Secure boot, DRM, mobile payments, key management, etc. |
| Vulnerabilities | Side-channel attacks, Spectre, Meltdown, etc. | Side-channel attacks, information leakage, etc. |

Both Intel SGX and ARM TrustZone are hardware-based security features designed to provide isolated execution environments for trusted code and data. They offer different mechanisms for isolation (enclaves vs. two worlds) and have

unique implementations in their respective processor architectures (x86-based for SGX and ARM-based for TrustZone). While these technologies enhance the security of sensitive operations, they are not immune to various attacks, and ongoing research and development is required to improve their security.

---

Question: `[13:58]` What are different between ARM SMC and Intel SMI? Please also summarize in a table. `(TEE.0328.md)`

Answer:

ARM SMC (Secure Monitor Call) and Intel SMI (System Management Interrupt) are both mechanisms used in their respective processor architectures for managing secure or privileged operations. Here's a summary of their differences in table format:

| Feature | ARM SMC | Intel SMI |
|---|---|---|
| Full Name | Secure Monitor Call | System Management Interrupt |
| Type | Instruction for switching between secure & non-secure worlds | Interrupt for managing system-level functions |
| Architecture | ARM-based processors | x86-based processors |
| Purpose | Transition between secure and non-secure worlds in ARM TrustZone | Execute system management functions in System Management Mode (SMM) |
| Invocation | Issued by software to request a switch between worlds | Triggered by hardware events or software requests |
| Execution Context | Secure world in ARM TrustZone | System Management Mode on x86 processors |
| Privilege Level | Highest privilege level in ARM TrustZone | Highest privilege level in x86 processors |
| Applications | Secure boot, DRM, mobile payments, key management, etc. | Power management, hardware configuration, firmware updates, etc. |

ARM SMC is an instruction used in ARM-based processors to switch between the secure and non-secure worlds of ARM TrustZone, enabling the processor to execute secure operations. In contrast, Intel SMI is an interrupt mechanism used in x86-based processors to manage system-level functions, such as power management and hardware configuration, in the System Management Mode (SMM). While both mechanisms operate at the highest privilege levels of their respective architectures, they serve different purposes and are invoked under different circumstances.

---

Question: `[13:59]` Please describe TEE in Intel. `(TEE.0328.md)`

Answer:

In Intel processors, the primary Trusted Execution Environment (TEE) technology is Intel Software Guard Extensions (SGX). Intel SGX provides a hardware-based TEE to securely execute sensitive code and protect data within isolated regions, known as enclaves.

Here's an overview of Intel SGX:

1. Enclaves: Enclaves are secure, isolated regions in the processor's memory that cannot be accessed by other processes or privileged software, such as the operating system, hypervisors, or other system software. They provide confidentiality and integrity for the data and code being executed within the enclave.
2. Memory encryption: Intel SGX encrypts the enclave memory using hardware-based encryption, which ensures that the data within the enclave remains confidential even if the physical memory is compromised.
3. Remote attestation: Intel SGX supports remote attestation, which enables a remote party to verify the integrity and authenticity of the enclave. This process ensures that the enclave is running the expected code and has not been tampered with.
4. Sealing: Intel SGX provides sealing functionality, which allows data to be encrypted and stored outside the enclave using

a key that is bound to the enclave's identity. This ensures that the data can only be decrypted by the same enclave or an enclave with the same identity, providing confidentiality for data at rest.

5. Application development: Developers can create applications that use Intel SGX by partitioning their code and data into trusted and untrusted components. The trusted components, which contain sensitive operations and data, are executed within enclaves, while the untrusted components are executed outside the enclave.

Intel SGX is a powerful TEE solution that provides an additional layer of security for sensitive code and data execution. However, it is important to note that Intel SGX is not immune to attacks, and researchers have discovered vulnerabilities in the technology, emphasizing the need for ongoing research and development to enhance its security.

---

Question: `[14:00]` What are APIs of TEE? Please classify them. `(TEE.0328.md)`

Answer:

Trusted Execution Environment (TEE) APIs are interfaces that enable developers to create applications that leverage the features of a TEE. They allow developers to interact with the secure environment and manage its resources, such as enclaves, secure storage, and cryptographic operations.

These APIs can be classified into several categories:

1. TEE Management APIs: - Enclave or secure world creation and destruction - Resource management, such as memory allocation and deallocation - Context switching between the secure and non-secure worlds
2. Cryptographic APIs: - Key management, including key generation, import, export, and storage - Cryptographic operations, such as encryption, decryption, signing, and verification - Secure random number generation - Cryptographic hash functions and message authentication codes (MACs)
3. Secure Storage APIs: - Secure storage of sensitive data, keys, and credentials - Data integrity and confidentiality protection - Access control and permission management
4. Attestation APIs: - Enclave or secure world measurement and reporting - Remote attestation to verify the integrity and authenticity of a TEE - Sealing and unsealing of data bound to the enclave or secure world identity
5. Communication APIs: - Secure communication channels between the TEE and external entities - Secure communication between enclaves or secure worlds

Examples of TEE API implementations include the GlobalPlatform TEE APIs, which provide a standardized set of APIs for TEEs, and platform-specific APIs like Intel SGX SDK for Intel Software Guard Extensions and ARM TrustZone APIs for ARM TrustZone-based TEEs.

It is important to note that the specific APIs available and their implementation may vary depending on the TEE technology used. However, the general categories mentioned above should apply to most TEE implementations.

---

Question: `[14:02]` Can I add a myself TEE API? `(TEE.0328.md)`

Answer:

Adding your own TEE API is possible but depends on the specific TEE implementation you are working with and the level of access and customization it allows. Some TEEs offer more flexibility for developers to extend and create their own APIs, while others may have more restrictive environments.

Here are a few steps to consider when adding your own TEE API:

1. Understand the TEE platform: Gain a deep understanding of the TEE platform you are working with, its architecture, and the existing APIs it provides. Familiarize yourself with the development environment, tools, and documentation.
2. Identify the use case: Determine the specific functionality or use case you want to address with your custom TEE API. Ensure that it is not already covered by existing APIs and that it aligns with the security objectives of the TEE.
3. Design the API: Design the API interface, considering factors such as function signatures, input and output parameters, and error handling. Keep security best practices in mind during the design process.

4. Implement the API: Write the code for your custom TEE API, following the platform's guidelines and using the development tools provided by the TEE vendor. Be sure to adhere to secure coding practices to avoid introducing vulnerabilities in your implementation.
5. Test and validate: Thoroughly test your custom TEE API, including edge cases and potential error conditions. Validate its functionality and security in the context of the TEE platform.
6. Integrate and document: Integrate your custom TEE API with your application or system, and create documentation to help other developers understand how to use your API.

Keep in mind that adding your own TEE API may introduce security risks if not done properly. It is crucial to ensure that your implementation adheres to security best practices and does not compromise the integrity and confidentiality of the TEE.

Before attempting to add your own TEE API, it is recommended to explore existing APIs and functionality provided by the TEE platform, as they are designed to address common use cases and have undergone extensive testing and validation.

---

## Memory `(230327.0P-TEE.yaml)`

Question: `[16:19]` How does OP-TEE manage memory? `(OP-TEE.0327.md)`

Answer:

OP-TEE manages memory within the secure world to ensure isolation, security, and efficient use of resources. Memory management in OP-TEE involves several aspects:

1. Secure memory allocation: OP-TEE uses secure memory for storing sensitive data and code related to the secure kernel and trusted applications (TAs). It ensures that secure memory is not accessible by the normal world to maintain confidentiality and integrity. OP-TEE provides APIs for trusted applications to allocate and deallocate secure memory dynamically.
2. Trusted Application memory isolation: OP-TEE maintains memory isolation between different trusted applications, preventing them from accessing each other's memory regions. This ensures the confidentiality and integrity of data and code in each trusted application.
3. Shared memory management: OP-TEE manages shared memory to facilitate secure data exchange between the normal world and the secure world. The GlobalPlatform TEE Client API provides functions to allocate, register, and release shared memory. OP-TEE ensures proper access control and isolation for shared memory buffers.
4. Memory mapping: OP-TEE manages the memory mapping for the secure world, ensuring that the memory regions for the secure kernel, trusted applications, and shared memory are correctly mapped to physical memory addresses.
5. Memory protection: OP-TEE enforces memory protection policies for the secure world, including access controls, permissions, and execution restrictions. This helps prevent unauthorized access, tampering, or execution of secure memory regions.
6. Memory management for trusted applications: Trusted applications in OP-TEE can request memory from the TEE Core, which is the secure kernel, through TEE Internal API functions like TEE_Malloc, TEE_Realloc, and TEE_Free. These functions allow trusted applications to allocate and manage memory dynamically within their secure environment.

In summary, OP-TEE manages memory within the secure world to ensure isolation, security, and efficient use of resources. This includes secure memory allocation, isolation between trusted applications, shared memory management, memory mapping, memory protection, and dynamic memory management for trusted applications.

---

Question: `[16:27]` Are data in memory in secure world encrypted? `(OP-TEE.0327.md)`

Answer:

Data stored in the memory of the secure world in OP-TEE is not encrypted by default. However, the secure world memory is protected by the hardware-based isolation provided by Arm TrustZone technology. TrustZone ensures that the normal world cannot directly access or tamper with the secure world's memory, which includes the secure kernel, trusted applications, and their associated data.

Although the data in the secure world memory is not encrypted, the isolation and access controls provided by TrustZone offer a strong level of protection. If an application or use case demands an even higher level of security, it is possible for trusted applications to implement additional encryption or other security measures for sensitive data stored in memory. This would depend on the specific requirements and threat model of the application or system.

In summary, data in the secure world memory is not encrypted by default in OP-TEE, but it is protected by the hardware-based isolation provided by Arm TrustZone technology. Additional encryption or security measures can be implemented by trusted applications if required.

---

## Signature `(230327.OP-TEE.yaml)`

Question: `[16:44]` Is signature required when a normal world application request a service in secure world? `(OP-TEE.0327.md)`

Answer:

A signature is not strictly required when a normal world application requests a service from a trusted application in the secure world. The communication between the normal world application and the trusted application is facilitated using the GlobalPlatform TEE Client API, which ensures proper access control and isolation for shared memory buffers used for data exchange.

However, a signature may be used in certain scenarios to enhance security. For example, if the normal world application sends sensitive data or a critical command to the trusted application, it might be useful to sign the data or command to ensure integrity and authenticity. This would help prevent tampering or replay attacks, especially in cases where the normal world might be compromised.

In such cases, the normal world application can sign the data or command using a private key, and the trusted application in the secure world can verify the signature using the corresponding public key. This ensures that the data or command has not been altered and that it originates from a trusted source.

In summary, a signature is not strictly required when a normal world application requests a service from a trusted application in the secure world. However, signatures can be used in certain scenarios to enhance security by ensuring the integrity and authenticity of data or commands sent to the secure world.

---

## CA & TA `(230328.TEE.yaml)`

Question: `[16:56]` How does an application in normal world request a service with shared memory of a trusted application in a secure world? Please describe the process with APIs. `(TEE.0328.md)`

Answer:

In a Trusted Execution Environment (TEE), a normal world application can request a service from a trusted application in the secure world using shared memory for data exchange. The process involves allocating a memory buffer that can be accessed by both the normal world and the secure world, allowing for efficient and secure communication between the two. Here's an outline of the process using the GlobalPlatform TEE APIs as an example:

1. Initialize context: The normal world application initializes a TEE context, which represents a connection to the TEE. In the GlobalPlatform TEE APIs, this is done using the `TEEC_InitializeContext` function.
2. Open session: The normal world application opens a session with the trusted application in the secure world. In the GlobalPlatform TEE APIs, this is done using the `TEEC_OpenSession` function.
3. Allocate shared memory: The normal world application allocates a shared memory buffer that can be accessed by both the normal world and the secure world. In the GlobalPlatform TEE APIs, this is done using the `TEEC_RegisterSharedMemory` function.
4. Prepare data: The normal world application writes the input data and any necessary parameters into the shared memory buffer.
5. Issue the call: The normal world application issues a call to the trusted application in the secure world to request the service, providing a reference to the shared memory buffer as part of the call. In the GlobalPlatform TEE APIs, this is

done using the `TEEC_InvokeCommand` function.

6. Process the request: The trusted application in the secure world processes the request, reads the input data from the shared memory buffer, performs the required operations, and writes the response back into the shared memory buffer.
7. Return to the normal world: Once the trusted application in the secure world has completed processing the request, control is returned to the normal world application.
8. Process the response: The normal world application reads the response from the shared memory buffer and processes it accordingly.
9. Release shared memory: After the response has been processed, the normal world application releases the shared memory buffer. In the GlobalPlatform TEE APIs, this is done using the `TEEC_ReleaseSharedMemory` function.
10. Close session: The normal world application closes the session with the trusted application in the secure world. In the GlobalPlatform TEE APIs, this is done using the `TEEC_CloseSession` function.
11. Finalize context: The normal world application finalizes the TEE context, releasing any resources associated with it. In the GlobalPlatform TEE APIs, this is done using the `TEEC_FinalizeContext` function.

Please note that the specific APIs and mechanisms for requesting a service using shared memory in the secure world may vary depending on the TEE implementation being used. The example provided above is based on the GlobalPlatform TEE APIs, which are widely used in the industry for TEE development. Other TEE implementations, such as Intel SGX or ARM TrustZone, may have different APIs and mechanisms for managing shared memory communication between the normal world and the secure world.

---

Question: `[17:10]` Which functions are used by TA (trusted application) in the above process? `(TEE.0328.md)`

Answer:

In the process described above, the Trusted Application (TA) in the secure world is responsible for handling requests from the normal world application and performing the requested operations. To do this, the TA needs to implement certain functions to interact with the TEE and the normal world application. The specific functions depend on the TEE implementation being used. Here, we'll use the GlobalPlatform TEE Internal APIs as an example:

1. TA_CreateEntryPoint: This function is called when the TA is first loaded by the TEE. It can be used to perform initial setup or resource allocation. The function is optional and may be left empty if no specific actions are required at this stage.
2. TA_DestroyEntryPoint: This function is called when the TA is unloaded by the TEE. It can be used to perform cleanup or resource deallocation. The function is optional and may be left empty if no specific actions are required at this stage.
3. TA_OpenSessionEntryPoint: This function is called when the normal world application opens a session with the TA. It can be used to perform session-specific setup, such as allocating session-specific resources or initializing session context. The function should return TEE_SUCCESS on success or an appropriate error code on failure.
4. TA_CloseSessionEntryPoint: This function is called when the normal world application closes a session with the TA. It can be used to perform session-specific cleanup, such as deallocating session-specific resources or clearing session context. The function is optional and may be left empty if no specific actions are required at this stage.
5. TA_InvokeCommandEntryPoint: This function is called when the normal world application issues a call to the TA to request a service. The function is responsible for processing the request, reading the input data from the shared memory buffer (if used), performing the required operations, and writing the response back into the shared memory buffer (if used). The function should return TEE_SUCCESS on success or an appropriate error code on failure.

The specific functions and APIs used by a Trusted Application (TA) may vary depending on the TEE implementation being used. The example provided above is based on the GlobalPlatform TEE Internal APIs, which are widely used in the industry for developing TAs in a TEE. Other TEE implementations, such as Intel SGX or ARM TrustZone, may have different functions and APIs for implementing TAs in the secure world.

---

Question: `[17:16]` Could you please draw a sequence diagram of TEEC_InitializeContext, TEEC_OpenSession, TEEC_RegisterSharedMemory, TEEC_InvokeCommand, TEEC_ReleaseSharedMemory, TEEC_CloseSession,

TEEC_FinalizeContext, TA_CreateEntryPoint, TA_DestroyEntryPoint, TA_OpenSessionEntryPoint, TA_CloseSessionEntryPoint, TA_InvokeCommandEntryPoint `(TEE.0328.md)`

Answer:

As a text-based AI language model, I cannot directly draw diagrams. However, I can describe the sequence of events and interactions between the normal world application and the trusted application in the secure world using the APIs mentioned in your question. You can use this description to create your own sequence diagram.

1. The normal world application calls `TEEC_InitializeContext` to initialize a TEE context.
2. The normal world application calls `TEEC_OpenSession` to open a session with the trusted application. a. The TEE loads the trusted application if not already loaded. b. The TEE calls `TA_CreateEntryPoint` in the trusted application for initialization.
3. The normal world application calls `TEEC_RegisterSharedMemory` to allocate a shared memory buffer.
4. The normal world application writes input data to the shared memory buffer.
5. The normal world application calls `TEEC_InvokeCommand`, passing a reference to the shared memory buffer. a. The TEE calls `TA_OpenSessionEntryPoint` in the trusted application to indicate a new session is opened. b. The TEE calls `TA_InvokeCommandEntryPoint` in the trusted application to process the request. i. The trusted application reads input data from the shared memory buffer. ii. The trusted application performs the requested operations. iii. The trusted application writes the response back to the shared memory buffer.
6. The normal world application reads the response from the shared memory buffer.
7. The normal world application calls `TEEC_ReleaseSharedMemory` to release the shared memory buffer.
8. The normal world application calls `TEEC_CloseSession` to close the session with the trusted application. a. The TEE calls `TA_CloseSessionEntryPoint` in the trusted application to indicate the session is closed.
9. The normal world application calls `TEEC_FinalizeContext` to finalize the TEE context. a. The TEE unloads the trusted application if no other sessions are open. b. The TEE calls `TA_DestroyEntryPoint` in the trusted application for cleanup.

You can use this sequence of events to create a diagram illustrating the interactions between the normal world application, the TEE system, and the trusted application in the secure world using the specified APIs.

---

Question: `[17:25]` Is the above sequence described in GlobalPlatform? `(TEE.0328.md)`

Answer:

Yes, the sequence of events and interactions between the normal world application and the trusted application in the secure world, as described in the previous response, is based on the GlobalPlatform TEE specifications. GlobalPlatform provides a set of standard APIs, including the TEE Client API (used by the normal world application) and the TEE Internal API (used by the trusted application), to facilitate secure communication and service invocation in a Trusted Execution Environment (TEE).

The GlobalPlatform TEE specifications can be found on the GlobalPlatform website, which provides detailed documentation on the APIs and their usage: https://www.globalplatform.org/specs-library/?search=tee

It's important to note that the specific APIs and mechanisms for requesting a service using shared memory in the secure world may vary depending on the TEE implementation being used. The example provided in the previous response is based on the GlobalPlatform TEE APIs, which are widely used in the industry for TEE development. Other TEE implementations, such as Intel SGX or ARM TrustZone, may have different APIs and mechanisms for managing shared memory communication between the normal world and the secure world.

---

Question: `[17:27]` Which document describe it? `(TEE.0328.md)`

Answer:

The sequence described in the previous responses is based on the GlobalPlatform TEE specifications, which are divided into two main documents:

1. GlobalPlatform TEE Client API Specification: This document describes the APIs used by the normal world application to interact with the TEE, such as `TEEC_InitializeContext`, `TEEC_OpenSession`, `TEEC_RegisterSharedMemory`, `TEEC_InvokeCommand`, `TEEC_ReleaseSharedMemory`, `TEEC_CloseSession`, and `TEEC_FinalizeContext`. The latest version of this document can be found at:

https://globalplatform.org/specs-library/tee-client-api-specification/

2. GlobalPlatform TEE Internal Core API Specification: This document describes the APIs used by the trusted application in the secure world, such as `TA_CreateEntryPoint`, `TA_DestroyEntryPoint`, `TA_OpenSessionEntryPoint`, `TA_CloseSessionEntryPoint`, and `TA_InvokeCommandEntryPoint`. The latest version of this document can be found at:

https://globalplatform.org/specs-library/tee-internal-core-api-specification/

Both documents provide detailed information on the APIs and their usage, including the sequence of interactions between the normal world application and the trusted application in the secure world. By studying these documents, you can gain a deep understanding of the TEE development process based on the GlobalPlatform specifications.

---

Question: `[17:36]` Please give me two sample codes of the above sequence. One is for CA and another is for TA. `(TEE.0328.md)`

Answer:

Here are two sample codes for the normal world application (Client Application, CA) and the secure world application (Trusted Application, TA) based on the GlobalPlatform TEE APIs:

Client Application (CA) - Normal World:

```
#include <stdio.h>
#include <string.h>
#include <tee_client_api.h>

#define TA_UUID {0x12345678, 0x1234, 0x1234, {0x12, 0x34, 0x56, 0x78, 0x90, 0xAB, 0xCD, 0xEF}}

void main() {
    TEEC_Result res;
    TEEC_Context ctx;
    TEEC_Session session;
    TEEC_SharedMemory sharedMem;
    TEEC_Operation operation;
    uint32_t returnOrigin;

    // Initialize context
    res = TEEC_InitializeContext(NULL, &ctx);
    if (res != TEEC_SUCCESS) {
        printf("Failed to initialize context\n");
        return;
    }

    // Open session
    res = TEEC_OpenSession(&ctx, &session, &TA_UUID, TEEC_LOGIN_PUBLIC, NULL, NULL, &returnOrigin);
    if (res != TEEC_SUCCESS) {
        printf("Failed to open session\n");
        TEEC_FinalizeContext(&ctx);
        return;
    }

    // Allocate shared memory
    sharedMem.size = sizeof(int) * 2;
    sharedMem.flags = TEEC_MEM_INPUT | TEEC_MEM_OUTPUT;
    res = TEEC_RegisterSharedMemory(&ctx, &sharedMem);
    if (res != TEEC_SUCCESS) {
        printf("Failed to allocate shared memory\n");
        TEEC_CloseSession(&session);
        TEEC_FinalizeContext(&ctx);
        return;
    }

    // Prepare operation
    operation.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_PARTIAL_INPUT, TEEC_MEMREF_PARTIAL_OUTPUT, TEEC_
    operation.params[0].memref.parent = &sharedMem;
    operation.params[0].memref.offset = 0;
    operation.params[0].memref.size = sizeof(int);
    operation.params[1].memref.parent = &sharedMem;
    operation.params[1].memref.offset = sizeof(int);
    operation.params[1].memref.size = sizeof(int);

    int inputData = 5;
    int outputData = 0;
    memcpy(sharedMem.buffer, &inputData, sizeof(int));

    // Invoke command
    res = TEEC_InvokeCommand(&session, 0x0000, &operation, &returnOrigin);
    if (res != TEEC_SUCCESS) {
        printf("Failed to invoke command\n");
    } else {
        memcpy(&outputData, sharedMem.buffer + sizeof(int), sizeof(int));
        printf("Input: %d, Output: %d\n", inputData, outputData);
    }

    // Release shared memory
    TEEC_ReleaseSharedMemory(&sharedMem);

    // Close session and finalize context
    TEEC_CloseSession(&session);
    TEEC_FinalizeContext(&ctx);
}
```

Trusted Application (TA) - Secure World:

```
#include <tee_internal_api.h>
#include <tee_internal_api_extensions.h>

TEE_Result TA_CreateEntryPoint(void) {
    return TEE_SUCCESS;
}

void TA_DestroyEntryPoint(void) {
}

TEE_Result TA_OpenSessionEntryPoint(uint32_t paramTypes, TEE_Param params[4], void **sessionContext) {
    return TEE_SUCCESS;
}

void TA_CloseSessionEntryPoint(void *sessionContext

---

**Question:** `[17:40]` Please write sample code of TA again. The code should be complete. `(TEE.0328.md)

**Answer:**

Apologies for the incomplete code earlier. Here's the complete sample code for the Trusted Application (T

```c
#include <tee_internal_api.h>
#include <tee_internal_api_extensions.h>

TEE_Result TA_CreateEntryPoint(void) {
    TEE_Result res = TEE_SUCCESS;
    // Perform initialization tasks, if needed
    return res;
}

void TA_DestroyEntryPoint(void) {
    // Perform cleanup tasks, if needed
}

TEE_Result TA_OpenSessionEntryPoint(uint32_t paramTypes, TEE_Param params[4], void **sessionContext) {
    TEE_Result res = TEE_SUCCESS;
    // Perform session-specific initialization tasks, if needed
    return res;
}

void TA_CloseSessionEntryPoint(void *sessionContext) {
    // Perform session-specific cleanup tasks, if needed
}

TEE_Result TA_InvokeCommandEntryPoint(void *sessionContext, uint32_t commandID, uint32_t paramTypes, TEE_
    TEE_Result res = TEE_ERROR_NOT_SUPPORTED;

    if (commandID == 0x0000) { // Sample command
        if (paramTypes != TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_INPUT, TEE_PARAM_TYPE_MEMREF_OUTPUT, TEE_
            return TEE_ERROR_BAD_PARAMETERS;
        }

        TEE_MemBuffer *inputBuffer = params[0].memref.buffer;
        TEE_MemBuffer *outputBuffer = params[1].memref.buffer;

        if (params[0].memref.size != sizeof(int) || params[1].memref.size != sizeof(int)) {
            return TEE_ERROR_BAD_PARAMETERS;
        }

        int inputData = 0;
        TEE_MemMove(&inputData, inputBuffer, sizeof(int));

        int outputData = inputData * 2;
        TEE_MemMove(outputBuffer, &outputData, sizeof(int));
```
```

```
        res = TEE_SUCCESS;
    }

    return res;
}
```

This sample Trusted Application code defines the required entry points for creating, destroying, opening, and closing a session. The `TA_InvokeCommandEntryPoint` function implements a simple command that reads an integer from the input buffer, multiplies it by 2, and writes the result back to the output buffer.

---