# 230328.TEE

Created: 2023-04-05 18:01:50

# Questions

- Basic Concepts
  - https://en.wikipedia.org/wiki/Trusted_execution_environment
  - Please summarize Intel SGX and ARM TrustZone in tables.
  - What are different between ARM SMC and Intel SMI? Please also summarize in a table.
  - Please describe TEE in Intel.
  - What are APIs of TEE? Please classify them.
  - Can I add a myself TEE API?
  - Is TEE a software?
  - Does GlobalPlatform contains Chinese Cryptographic Standard?
- CA & TA
  - 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.
  - Which functions are used by TA (trusted application) in the above process?
  - 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
  - Is the above sequence described in GlobalPlatform?
  - Which document describe it?
  - Please give me two sample codes of the above sequence. One is for CA and another is for TA.
  - Please write sample code of TA again. The code should be complete.
  - Can CA and TA be implemented in other languages?
  - How does TA authenticate request from CA?
  - What are kinds of TA?

---

# Q & A

## Basic Concepts

Question: https://en.wikipedia.org/wiki/Trusted_execution_environment

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: Please summarize Intel SGX and ARM TrustZone in tables.

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: What are different between ARM SMC and Intel SMI? Please also summarize in a table.

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: Please describe TEE in Intel.

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: What are APIs of TEE? Please classify them.

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: Can I add a myself TEE API?

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.

---

Question: Is TEE a software?

Answer:

A Trusted Execution Environment (TEE) is not just a software; it is a combination of hardware, software, and firmware components that work together to provide a secure and isolated environment for executing sensitive operations. The TEE aims to protect sensitive data and code from unauthorized access, tampering, or leakage, even if the main operating system is compromised.

The main components of a TEE include:

1. Hardware: The hardware provides the necessary features to create an isolated environment that is separate from the main processor or the "Normal World." This can include dedicated CPU modes, memory protection mechanisms, and secure storage. Examples of hardware-based TEE implementations are ARM TrustZone and Intel Software Guard Extensions (SGX).
2. Software: The TEE software includes a secure operating system (often called a "Trusted OS" or "Secure OS") and a set of APIs that enable developers to create and interact with Trusted Applications (TAs). The software may also include additional security components, such as secure boot, cryptographic libraries, and attestation mechanisms. Examples of TEE software include OP-TEE for ARM TrustZone and Intel SGX SDK.
3. Firmware: Firmware components can also play a role in the TEE by providing low-level support for the hardware and software components. Firmware can be involved in the secure boot process, managing the lifecycle of the TEE, or providing cryptographic services.

In summary, a TEE is a combination of hardware, software, and firmware components designed to provide a secure environment for executing sensitive operations. It is not merely a software solution but an integrated system that leverages various layers of the computing stack to ensure security and isolation.

---

Question: Does GlobalPlatform contains Chinese Cryptographic Standard?

Answer:

GlobalPlatform does not specifically include Chinese Cryptographic Standards as part of its core specifications. GlobalPlatform provides a standardized framework for implementing secure elements, TEEs (Trusted Execution Environments), and SE (Secure Element) management. It focuses on defining the APIs, protocols, and security requirements for these environments, but it does not mandate specific cryptographic algorithms.

However, GlobalPlatform's specifications are designed to be flexible and extensible, allowing individual TEE or secure element implementations to support various cryptographic algorithms, including Chinese Cryptographic Standards, as required by the specific use case or market requirements.

For example, a TEE implementation targeting the Chinese market may choose to integrate Chinese Cryptographic Standards such as:

1. SM2: An elliptic curve cryptography (ECC) standard for digital signatures, key exchange, and public key encryption.
2. SM3: A cryptographic hash function with a 256-bit digest.
3. SM4: A symmetric encryption algorithm with a 128-bit block size and a 128-bit key size.

These algorithms can be integrated into the TEE or secure element implementation, either as built-in functions or through third-party cryptographic libraries. As long as the implementation adheres to GlobalPlatform's specifications regarding the APIs, protocols, and security requirements, it can support a wide range of cryptographic algorithms, including Chinese Cryptographic Standards.

---

## CA & TA

Question: 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.

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: Which functions are used by TA (trusted application) in the above process?

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: 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

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: Is the above sequence described in GlobalPlatform?

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: Which document describe it?

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: Please give me two sample codes of the above sequence. One is for CA and another is for TA.

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:

```c
#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: Please write sample code of TA again. The code should be complete.

Answer:

Apologies for the incomplete code earlier. Here’s the complete sample code for the Trusted Application (TA) in the secure world based on the GlobalPlatform TEE Internal API:

```
#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.

---

Question: Can CA and TA be implemented in other languages?

Answer:

Yes, both the Client Application (CA) and the Trusted Application (TA) can be implemented in other programming languages, as long as the language supports interfacing with the required TEE APIs. Many TEE implementations provide APIs and language bindings for multiple programming languages.

For instance, the GlobalPlatform TEE Client API (for the CA) is generally available as a C API, but you can create language bindings for other languages like Python, Java, or Rust. You might need to use the Foreign Function Interface (FFI) or a similar mechanism to interface with the C API from another language.

As for the Trusted Application (TA), the choice of programming languages may be more limited, depending on the TEE implementation and the underlying hardware. Some TAs are typically written in C or C++ for performance and low-level access to hardware features, but there are implementations that support other languages like Rust.

For example, the Rust programming language has gained traction in TEE development, as it provides strong safety guarantees and is suitable for systems programming. There are projects like Teaclave SGX SDK (https://github.com/apache/teaclave-sgx-sdk) for Intel SGX and Rust OP-TEE TrustZone SDK (https://github.com/OP-TEE/optee-rust) for ARM TrustZone, which enable developers to write TAs in Rust.

In summary, while the choice of programming languages for CA and TA might be determined by the specific TEE implementation and hardware, it is possible to use languages other than C or C++ for both normal world and secure world applications.

---

Question: How does TA authenticate request from CA?

Answer:

Trusted Applications (TAs) can authenticate requests from Client Applications (CAs) using a variety of mechanisms. One common approach is to establish a secure session between the CA and the TA, which can involve authentication, key exchange, and secure communication. This ensures that only authorized CAs can access the services provided by the TA.

Here are some methods that can be used to authenticate requests from a CA to a TA:

1. Login Methods: Different TEE implementations support various login methods to authenticate the origin of the request. GlobalPlatform TEE, for instance, supports multiple login types such as `TEEC_LOGIN_PUBLIC`, `TEEC_LOGIN_USER`, `TEEC_LOGIN_GROUP`, and `TEEC_LOGIN_APPLICATION`. These login types define different levels of trust and can be used to authenticate the CA based on the user, group, or application identity.
2. Client Authentication: When a CA opens a session with a TA, it can provide authentication data (e.g., a certificate, a pre-shared key, or a signature) as part of the session's parameters. The TA can verify this authentication data to ensure the CA is authorized to access its services. This can be implemented using custom or standardized cryptographic protocols.
3. Secure Channels: The CA and TA can establish a secure communication channel, which provides message confidentiality, integrity, and optionally, replay protection. This can be done by performing a key exchange and using shared secrets to encrypt and authenticate the communication between the CA and the TA.
4. Attestation: In some cases, the CA and/or the TA may require additional trust guarantees from the TEE or the platform. This can be achieved through attestation mechanisms, where a trusted third party (e.g., the TEE vendor or the platform owner) vouches for the authenticity and integrity of the TEE, the TA, or the CA. Examples of attestation mechanisms include Intel SGX's remote attestation and ARM TrustZone's secure boot.

The specific method of authentication and the level of trust required between the CA and the TA depend on the use case, the TEE implementation, and the underlying hardware. To ensure the highest level of security, it is essential to choose appropriate authentication mechanisms and follow best practices in implementing them.

---

Question: What are kinds of TA?

Answer:

Trusted Applications (TAs) can be broadly classified into two main categories based on their purpose and level of trust:

1. Rich Trusted Applications (RTAs): Rich Trusted Applications are TAs that provide more complex services and often interact with external components, such as peripherals, network services, or other TAs. RTAs are typically used for high-level security services like data encryption, digital rights management (DRM), secure payment processing, or secure authentication. These applications might run in a more feature-rich environment with support for a wider range of APIs, libraries, and services. RTAs generally have higher privileges and access to more resources than LTAs.
2. Lightweight Trusted Applications (LTAs): Lightweight Trusted Applications are smaller, more focused TAs that provide specific security services. These applications usually have limited functionality and are designed for low-latency, low-power, or resource-constrained environments. LTAs typically have a smaller Trusted Computing Base (TCB), which reduces the attack surface and increases the security of the TA. Examples of LTAs include secure key storage, cryptographic operations, and basic device attestation.

Additionally, TAs can be classified based on their development model and level of trust:

1. Vendor Trusted Applications: These are TAs developed by the TEE vendor or the device manufacturer. They are often pre-installed on the device and have the highest level of trust. Vendor TAs are typically responsible for managing the TEE, providing basic security services, or implementing device-specific features.
2. Third-Party Trusted Applications: These are TAs developed by third-party developers, such as independent software vendors (ISVs) or enterprise developers. Third-party TAs usually provide additional security services or features that are not covered by the vendor TAs. The trust level of third-party TAs depends on the authentication and attestation mechanisms used by the TEE and the device.

In summary, TAs can be classified based on their purpose, complexity, level of trust, and development model. The specific types of TAs supported by a TEE may vary depending on the TEE implementation, the underlying hardware, and the security requirements of the device.