230327.OP-TEE

Created: 2023-04-05 20:10:51

Questions

- Basic Concepts
  - https://github.com/OP-TEE
  - What is OP-TEE?
  - What is thre prefix OP of OP-TEE?
  - Are there alternative technologies of OP-TEE?
  - Does OP-TEE a software or a hardware solution?
  - Is OP-TEE a firmware?
  - Does OP-TEE only exist in secure world?
  - Are there successfull cases of OP-TEE?
  - What are different between ARM OP-TEE and Intel SMM? Please also summarize in a table.
- Boot
  - Does OP-TEE OS exist in a single core or multiple cores at run-time?
  - Why does OP-TEE's reset function have two branches reset_primary and reset_secondary?
  - Why does OP-TEE initialize secondary cores when reset?
- Multi-Thread & Multi-Process
  - Can secure kernel and trusted applications in OP-TEE can be multi-thread?
  - What is secure context switching in TrustZone?
  - Does OP-TEE support multi-process?
  - Does OP-TEE support IPC?
- Memory
  - Does OP-TEE support shared memory?
  - Can secure world can access memory in normal world?
  - Can normal world access memory in secure world?
  - How does OP-TEE manage memory?
  - Are data in memory in secure world encrypted?
  - What is SMAF?
  - What is CMA?
- Network
  - Does OP-TEE support network communication in secure world?
- Signature
  - Is signature required when a normal world application request a service in secure world?
  - Does OP-TEE support to verify a signature of a TA in X.509 certificate?
- Development
  - Which languages do I use to develop apps in OP-TEE or to customize OP-TEE?
  - Can I use other langauges?
  - Does OP-TEE support ARMv7 and ARMv8?
  - Does OP-TEE have a shell?
  - Can I create a TA at run-time?
  - But what is TEECI_LoadSecureModule()
  - What are Static TA and Dynamic TA? Please also summarize them in a table.
  - Does TEE define Static TA and Dynamic TA?
  - How to recover a crashed static TA?
  - How to recover a crashed dynamic TA?
- Qualcomm

    - Does Qualcomm support OP-TEE?
    - Is QTEE open source?
- DRM
    - How do I implement DRM of content decryption with OP-TEE?
    - So the content is decrypted in TA in secure world. Right?
    - But does it slow down the playback speed because of switching between normal world and secure world?
    - Can I use multi cores to solve the problem? For example, one core decrypts content and another core play content at the same time.
    - Does OP-TEE support TUI?
    - Can I develop a TA with TUI to play video in OP-TEE?
    - How do I implement a DRM solution using ClearKey in OP-TEE?
- Secure Storage
    - Does OP-TEE support file system?
    - How does OP-TEE generate SSK from HUK in tee_fs_init_key_manager()?
    - In the function, can I read the content of the HUK?
    - How do I protect HUK from reading in OP-TEE?
    - Is SSK defined in GlobalPlatform?
- Recover OP-TEE
    - How to detect OP-TEE dead?
    - How to automatically recovery the crashed OP-TEE?
    - Can I develop a dual OP-TEE so that if one is crashed, run another.
    - How to detect if OP-TEE is tamperred at run-time?
- Attack
    - Whare are kinds of methods to attack OP-TEE?

Q & A

Basic Concepts

Question: https://github.com/OP-TEE

Answer:

The link you provided, https://github.com/OP-TEE, is the official GitHub repository for the OP-TEE project. This repository contains the source code, documentation, and other resources related to OP-TEE, an open-source Trusted Execution Environment (TEE) based on the Arm TrustZone technology.

The repository is organized into several sub-projects, including:

1. optee_os: The core component, which includes the secure kernel and related components that run within the secure world. It manages trusted applications, resources, and secure context switching.

2. optee_client: The library implementing the GlobalPlatform TEE Client API, which enables normal world applications to communicate with trusted applications in the secure world.

3. optee_test: A set of tests and examples to verify the functionality of the OP-TEE system, including trusted application examples and test suites for various cryptographic operations.

4. build: A set of scripts and configuration files to help users build and compile the OP-TEE components for different target platforms.

The GitHub repository is an excellent starting point for developers interested in working with OP-TEE, as it provides the necessary resources and documentation to understand, build, and deploy OP-TEE on supported platforms. If you are interested in learning more or contributing to the project, you can explore the repository, read the documentation, and follow the guidelines provided.

Question: What is OP-TEE?

Answer:

OP-TEE (Open Portable Trusted Execution Environment) is an open-source project that aims to provide a secure and trusted execution environment for software applications on devices. OP-TEE is primarily designed for Arm TrustZone technology, which is used in various processors to create a secure area (trusted execution environment) separate from the normal operating system. This separation helps protect sensitive data and code from unauthorized access or tampering.

OP-TEE is made up of several components, including:

1. Secure kernel: This is the core component of OP-TEE that runs in the secure world and manages trusted applications (TAs) and their execution.
2. Trusted applications: These are the secure applications that run within the secure world, providing security services to normal applications.
3. Client API: This is the interface that normal applications use to communicate with trusted applications.
4. TEE Internal Core API: This is the interface used by trusted applications to access TEE services provided by the secure kernel.

OP-TEE is widely used in industries like mobile devices, IoT, automotive, and more, where security is of paramount importance. The project is maintained by Linaro, a collaborative organization that focuses on improving open-source software for Arm-based platforms.

Question: What is thre prefix OP of OP-TEE?

Answer:

The prefix “OP” in OP-TEE stands for “Open Portable.” It signifies that the project is an open-source implementation of a Trusted Execution Environment (TEE) that can be portably used across various platforms, mainly built around Arm TrustZone technology. The name highlights the goals of providing both transparency and flexibility to developers and users who want to create secure applications and services on a wide range of devices.

Question: Are there alternative technologies of OP-TEE?

Answer:

Yes, there are alternative technologies to OP-TEE that provide trusted execution environments or similar security features. Some of them include:

1. Intel SGX (Software Guard Extensions): Intel SGX is a set of security-related instruction codes that are built into some Intel CPUs. It allows developers to create secure enclaves, which are isolated memory regions that protect sensitive data and code from unauthorized access, even from the host operating system or a hypervisor.

2. AMD SEV (Secure Encrypted Virtualization): AMD SEV is a hardware feature found in some AMD processors that encrypts the memory of virtual machines to protect data from unauthorized access. It aims to provide a secure execution environment for applications running in virtualized environments.

3. ARM TrustZone-M: TrustZone-M is a security extension for ARM Cortex-M processors, targeting microcontroller devices used in IoT, industrial control, and other embedded systems. While OP-TEE targets TrustZone technology in application processors, TrustZone-M is tailored for microcontrollers.

4. Google Titan M: Titan M is a custom security chip developed by Google to provide a secure environment for storing sensitive data and performing cryptographic operations on its Pixel smartphones. It is designed to protect the integrity of the bootloader, verify the operating system, and secure user data.

5. Apple Secure Enclave: The Secure Enclave is a coprocessor found in Apple devices, such as iPhones and iPads, that provides a secure environment for cryptographic operations and storage of sensitive data like encryption keys, biometric information, and device passcodes.

These technologies may have different implementations, features, and target applications, but they all aim to enhance security by providing isolated execution environments and protecting sensitive data from unauthorized access.

Question: Does OP-TEE a software or a hardware solution?

Answer:

OP-TEE is primarily a software solution that leverages underlying hardware capabilities, specifically Arm TrustZone technology, to provide a secure and trusted execution environment for software applications on devices.

OP-TEE includes a secure kernel, trusted applications, and APIs that enable secure communication between the normal and secure worlds. While OP-TEE is a software implementation, it relies on the hardware-based security features provided by Arm TrustZone to create a separate, secure area for sensitive data and code execution.

In summary, OP-TEE is a software solution that takes advantage of hardware-based security technologies to achieve its goals of providing a secure environment for running trusted applications.

Question: Is OP-TEE a firmware?

Answer:

OP-TEE can be considered a firmware component, as it is a low-level software that interacts directly with the hardware and provides security services to the system. It serves as the trusted execution environment for devices using Arm TrustZone technology.

Firmware is a type of software that is typically stored in non-volatile memory and provides basic functionality or services to a device’s hardware. In the case of OP-TEE, it provides the secure kernel, trusted applications, and APIs for managing and interacting with the secure environment.

OP-TEE is often integrated into the boot process of a device, alongside other firmware components like the bootloader, and it initializes the secure environment before the main operating system starts. This ensures that the secure services and trusted applications provided by OP-TEE are available to the system throughout its operation.

Question: Does OP-TEE only exist in secure world?

Answer:

OP-TEE is a Trusted Execution Environment (TEE) that leverages Arm TrustZone technology, which provides hardware-based isolation between the normal world and the secure world. OP-TEE consists of components that exist in both the normal world and the secure world.

1. Secure world components:

- Secure kernel: The OP-TEE secure kernel resides in the secure world and manages trusted applications, secure memory, and communication between the normal world and the secure world.
- Trusted applications: These are secure services or applications that run in the secure world and provide functionality that requires a higher level of security and isolation.

2. Normal world components:

- TEE Client API: This is a library that normal world applications use to communicate with the secure world. It provides an interface for opening sessions, invoking commands, and managing shared memory with trusted applications.
- TEE supplicant: The TEE supplicant is a user-space daemon in the normal world that handles certain requests from the secure world, such as loading trusted applications, filesystem operations, and cryptographic operations that involve the normal world’s resources.

Both the secure world and normal world components of OP-TEE work together to provide a secure and isolated environment for trusted applications, as well as a communication channel between normal world applications and the secure world.

In summary, OP-TEE does not only exist in the secure world. It consists of components in both the normal world and the secure world to create a comprehensive TEE solution that leverages Arm TrustZone technology.

Question: Are there successfull cases of OP-TEE?

Answer:

Yes, there have been successful use cases of OP-TEE across various industries and applications that require a secure and trusted execution environment. Some of these include:

1. Mobile devices: OP-TEE is used in mobile devices like smartphones and tablets to provide a secure environment for handling sensitive data, such as biometric information, cryptographic keys, and DRM (Digital Rights Management) protection.
2. IoT devices: Internet of Things (IoT) devices, such as smart home appliances, wearables, and industrial sensors, benefit from the security provided by OP-TEE. It enables these devices to handle sensitive data securely, execute cryptographic operations, and maintain the confidentiality and integrity of the data being transmitted.
3. Automotive systems: OP-TEE can be used in automotive systems to secure in-vehicle communication, protect sensitive data, and enable secure over-the-air updates.
4. Payment systems: OP-TEE can provide a secure environment for payment applications, ensuring the confidentiality and integrity of financial transactions and sensitive user data.
5. Healthcare devices: Medical devices and healthcare systems can utilize OP-TEE to securely store and process patient data, protecting it from unauthorized access and tampering.

While specific implementations may vary, these examples demonstrate the versatility and effectiveness of OP-TEE in providing a secure and trusted environment for a wide range of applications. The open-source nature of OP-TEE also makes it possible for developers and organizations to customize and adapt the software to fit their unique security requirements.

Question: What are different between ARM OP-TEE and Intel SMM? Please also summarize in a table.

Answer:

ARM OP-TEE and Intel SMM are different security technologies that provide secure and isolated execution environments for sensitive operations. Here’s a comparison of the two:

1. ARM OP-TEE:

- Primarily designed for ARM-based platforms using TrustZone technology.
- Open-source implementation, maintained by Linaro.
- Provides a secure kernel, trusted applications, and APIs for communication between the normal and secure worlds.
- Focuses on creating a Trusted Execution Environment (TEE) for secure application execution.
- Targets a variety of industries, including mobile devices, IoT, automotive systems, and more.

2. Intel SMM (System Management Mode):

- Designed for Intel x86 processors.
- A proprietary implementation by Intel.
- Provides a separate CPU mode (SMM) with its own address space and execution environment.
- Focuses on handling low-level system management tasks, such as power management, hardware monitoring, and firmware updates.
- Primarily used for system management tasks on PCs, servers, and other x86-based systems.

Summary in a table:

Feature/Aspect	ARM OP-TEE	Intel SMM
Platform	ARM-based systems	Intel x86 processors
Implementation	Open-source	Proprietary
Primary Focus	Trusted Execution Environment (TEE)	System Management Tasks
Typical Applications	Mobile, IoT, automotive, etc.	PCs, servers, x86-based systems
Architecture	ARM TrustZone	System Management Mode

Please note that while both technologies provide a secure and isolated execution environment, they serve different purposes and target different platforms. ARM OP-TEE is designed to enable secure applications on ARM-based devices, while Intel SMM is used for low-level system management tasks on Intel x86-based systems.

Boot

Question: Does OP-TEE OS exist in a single core or multiple cores at run-time?

Answer:

At runtime, the OP-TEE OS exists in a single core. It is designed to run on one core at a time in the secure world. However, OP-TEE supports concurrent execution of Trusted Applications (TAs) on different CPU cores. Although the OS itself runs on a single core, it can manage and schedule TAs running on separate cores, ensuring proper execution and isolation.

Question: Why does OP-TEE's reset function have two branches reset_primary and reset_secondary?

Answer:

In OP-TEE, the reset function has two branches, `reset_primary` and `reset_secondary`, to handle the initialization process for multi-core systems. These branches are responsible for initializing the primary and secondary CPU cores, respectively, within the OP-TEE secure world.

1. `reset_primary`: This branch handles the initialization of the primary CPU core (core 0). The primary core is responsible for setting up the secure environment, initializing the secure kernel, and starting the other secondary cores. The primary core performs tasks such as configuring the MMU (Memory Management Unit), initializing the secure memory, setting up the interrupt controller, and more. Once the primary core has completed its initialization tasks, it starts executing the secure kernel.

2. `reset_secondary`: This branch handles the initialization of secondary CPU cores (core 1, core 2, etc.). The secondary cores also need to set up their own environment, such as configuring their local MMU and enabling their local interrupts, before they can start executing the secure kernel. The primary core signals the secondary cores to begin their initialization process after it has completed its own initialization.

The distinction between `reset_primary` and `reset_secondary` branches ensures that the primary core properly initializes the secure environment before the secondary cores start executing the secure kernel. This separation helps maintain a correct and secure boot process in multi-core systems, where multiple CPU cores need to work together within the secure world.

In summary, OP-TEE's reset function has two branches, `reset_primary` and `reset_secondary`, to handle the initialization process for multi-core systems, ensuring that the primary core sets up the secure environment correctly before the secondary cores start executing the secure kernel.

Question: Why does OP-TEE initialize secondary cores when reset?

Answer:

OP-TEE initializes secondary cores during the reset process to properly configure and synchronize them with the primary core. This is important for maintaining a consistent and secure execution environment across all cores.

When the system is reset, the primary core (core 0) is responsible for initializing the OP-TEE OS, including the secure world and shared resources. Once the primary core has completed the initialization process, it sends a signal to secondary cores to begin their own initialization process. Each secondary core then initializes its own secure world environment and sets up necessary data structures and configurations in alignment with the primary core.

This process ensures that all cores have a consistent view of the secure world, enabling the execution of Trusted Applications (TAs) across multiple cores while maintaining security and isolation.

Multi-Thread & Multi-Process

Question: Can secure kernel and trusted applications in OP-TEE can be multi-thread?

Answer:

As of now, the secure kernel and trusted applications in OP-TEE are primarily single-threaded. The secure world in OP-TEE does not support multi-threading within a single trusted application, meaning each trusted application can only execute one thread at a time.

However, OP-TEE does support parallelism in the sense that it allows multiple trusted applications to be loaded and executed concurrently. This is achieved through the use of secure context switching between different trusted applications, ensuring that only one trusted application is active at a time.

Although the secure environment itself is single-threaded, OP-TEE still enables concurrent communication with normal world applications. Multiple normal world applications can communicate with different trusted applications simultaneously, and OP-TEE will manage the context switching between them.

In summary, the secure kernel and trusted applications in OP-TEE are primarily single-threaded, but parallelism is achieved through secure context switching between multiple trusted applications and concurrent communication with normal world applications.

Question: What is secure context switching in TrustZone?

Answer:

Secure context switching in TrustZone refers to the process of transitioning between different secure contexts (trusted applications) within the secure world while maintaining isolation and security. ARM TrustZone technology provides a hardware-based separation between the normal world and the secure world, allowing both to run independently and securely.

In the context of OP-TEE and TrustZone, when multiple trusted applications are running or when a normal world application requests a service from a trusted application, the secure kernel manages the execution of these trusted applications by performing secure context switching. This involves:

1. Saving the current context: The secure kernel saves the current state of the active trusted application, including its register values and memory, in a secure location.
2. Restoring the new context: The secure kernel restores the state of the trusted application that is being switched to by loading its previously saved register values and memory.
3. Switching execution: The secure kernel transfers control to the new trusted application, allowing it to continue executing where it left off.

Secure context switching ensures that each trusted application’s state is kept separate and secure from others, maintaining isolation and preventing unauthorized access or tampering. Additionally, it allows the secure kernel to manage the concurrent execution of multiple trusted applications and handle requests from normal world applications efficiently.

In summary, secure context switching in TrustZone is the process of managing and transitioning between different trusted applications within the secure world while maintaining isolation, security, and efficient execution.

Question: Does OP-TEE support multi-process?

Answer:

OP-TEE supports multiple trusted applications (TAs) running concurrently within the secure environment provided by the Arm TrustZone. While the secure world in OP-TEE is primarily single-threaded, it can handle multiple processes in the form of trusted applications.

When multiple trusted applications are loaded and running, OP-TEE manages their execution using secure context switching. This ensures that only one trusted application is active at a time, while maintaining isolation and security between different trusted applications. The secure kernel in OP-TEE is responsible for managing resources, handling requests from the normal world, and switching between the trusted applications as needed.

On the normal world side, multiple processes can communicate with different trusted applications simultaneously, and OP-TEE will manage the context switching between them.

In summary, OP-TEE supports multi-process execution in the form of multiple trusted applications running concurrently within the secure environment, while also allowing multiple normal world processes to communicate with these trusted applications.

Question: Does OP-TEE support IPC?

Answer:

Yes, OP-TEE supports Inter-Process Communication (IPC) between trusted applications (TAs) within the secure world and between normal world applications and trusted applications.

1. IPC between trusted applications: Trusted applications in OP-TEE can share data and communicate with each other by invoking other trusted applications. However, the secure kernel must intermediate such communication to ensure isolation and maintain security. Trusted applications use the TEE_InvokeTACommand function to invoke another trusted application and pass data as parameters.

2. IPC between normal world applications and trusted applications: OP-TEE supports communication between normal world applications and trusted applications through the GlobalPlatform TEE Client API. Normal world applications can open a session with a trusted application, invoke commands, and share data with the trusted application while maintaining security.

In both cases, OP-TEE manages the IPC to ensure isolation between different trusted applications and maintain the security of the secure environment. While the communication mechanisms may vary, OP-TEE provides the necessary APIs and infrastructure to support IPC between trusted applications and between normal world applications and trusted applications.

Memory

Question: Does OP-TEE support shared memory?

Answer:

Yes, OP-TEE supports shared memory, which allows normal world applications to share data with trusted applications (TAs) within the secure environment. Shared memory is used for exchanging data between the normal world and the secure world while maintaining security and isolation.

In OP-TEE, shared memory is managed by the GlobalPlatform TEE Client API, which provides functions for allocating, registering, and releasing shared memory. When a normal world application wants to share data with a trusted application, it creates a shared memory buffer using these API functions. The shared memory buffer is then passed as a parameter to the trusted application when invoking a command.

The secure kernel in OP-TEE ensures that the shared memory buffer is accessible by both the normal world application and the trusted application while preventing unauthorized access or tampering. The secure kernel also maintains the isolation between different trusted applications by limiting access to the shared memory buffers associated with each trusted application.

Please note that shared memory in OP-TEE is primarily used for communication between normal world applications and trusted applications. Sharing memory directly between different trusted applications is generally not supported, as it can compromise the isolation and security provided by the secure environment. Instead, trusted applications communicate with each other using the TEE_InvokeTACommand function, as mentioned in a previous answer regarding IPC support.

In summary, OP-TEE supports shared memory for exchanging data between normal world applications and trusted applications while maintaining security and isolation within the secure environment.

Question: Can secure world can access memory in normal world?

Answer:

In general, the secure world in OP-TEE can access memory in the normal world, but it must do so with caution. The secure world has a higher level of privilege, allowing it to access both secure and normal world memory. However, accessing normal world memory from the secure world can introduce potential security risks, such as data leaks or manipulation of sensitive information.

To minimize such risks, OP-TEE employs a set of precautions and guidelines:

1. Use shared memory: When secure world applications need to access data from the normal world, it is recommended to use shared memory, which is explicitly allocated and managed by the GlobalPlatform TEE Client API. This ensures proper isolation and access control for the shared data.

2. Validate memory addresses: If the secure world must access normal world memory directly, it should first validate the memory address range to ensure it doesn't overlap with secure memory or any other protected memory region.

3. Be cautious of data manipulation: The secure world should be aware that normal world memory can be manipulated or tampered with by potentially malicious software running in the normal world. Sensitive operations should not rely solely on data stored in normal world memory without additional validation or protection.

4. Minimize normal world memory access: The secure world should minimize direct access to normal world memory whenever possible and instead rely on secure memory or shared memory buffers for data exchange.

While the secure world in OP-TEE can access normal world memory, it should be done with caution and following proper guidelines to maintain the security and isolation provided by the ARM TrustZone technology.

Question: Can normal world access memory in secure world?

Answer:

In the context of OP-TEE and Arm TrustZone technology, the normal world cannot directly access the memory in the secure world. The TrustZone technology enforces hardware-based isolation between the normal and secure worlds, which prevents normal world software from accessing or tampering with secure world memory.

The secure world memory, which includes the secure kernel, trusted applications, and their associated data, is protected from unauthorized access by the normal world. This ensures that sensitive data and operations within the secure world remain confidential and secure.

If a normal world application needs to exchange data with a trusted application in the secure world, it must use shared memory, which is explicitly allocated and managed by the GlobalPlatform TEE Client API. This provides a controlled mechanism for data exchange between the normal and secure worlds, maintaining security and isolation.

In summary, the normal world cannot directly access memory in the secure world, thanks to the hardware-based isolation provided by Arm TrustZone technology. Data exchange between the two worlds is facilitated through shared memory, which is managed by the GlobalPlatform TEE Client API.

Question: How does OP-TEE manage memory?

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: Are data in memory in secure world encrypted?

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.

Question: What is SMAF?

Answer:

SMAF stands for Secure Memory Allocation Framework, which is a part of the OP-TEE project. SMAF is designed to provide secure memory allocation and management in a Trusted Execution Environment (TEE). It allows OP-TEE to allocate, deallocate, and manage memory resources efficiently and securely within the secure world.

SMAF provides various features, including:

1. Secure memory allocation: SMAF enables the allocation of secure memory for Trusted Applications (TAs) and the OP-TEE OS within the secure world.
2. Memory isolation: SMAF ensures that memory allocated to a TA is isolated from other TAs and the OP-TEE OS, protecting sensitive data from unauthorized access or tampering.
3. Fine-grained access control: SMAF provides fine-grained access control over memory regions, allowing TAs and the OP-TEE OS to specify access permissions for different memory areas.

4. Efficient memory management: SMAF includes mechanisms for efficient memory management, such as memory pooling and caching, to optimize resource utilization and reduce memory fragmentation.

Overall, SMAF plays a critical role in maintaining the security and integrity of data within the OP-TEE environment. By providing secure memory allocation and management features, SMAF helps ensure that sensitive information remains protected from potential threats in the normal world and other trusted applications running in the secure world.

Question: What is CMA?

Answer:

CMA stands for Contiguous Memory Allocator, a feature in the Linux kernel that helps to allocate large contiguous blocks of memory. In systems where memory fragmentation is an issue, CMA can be particularly useful for devices and drivers that require large contiguous memory regions for their operation, such as multimedia hardware or devices using Direct Memory Access (DMA).

CMA works by reserving a portion of memory during the system boot process. This reserved memory region, known as the CMA pool, is kept separate from the general-purpose system memory. When a device or driver needs a large contiguous block of memory, it can request the allocation from the CMA pool. This approach reduces the likelihood of memory fragmentation and ensures that large contiguous memory regions are available when required.

CMA is not specific to OP-TEE but can be used in conjunction with OP-TEE in some cases. For example, OP-TEE may use CMA to allocate a large block of memory for a Trusted Application (TA) that requires a significant amount of contiguous memory. Additionally, CMA can be used to manage memory shared between the normal world and the secure world, ensuring that the shared memory regions are allocated contiguously and can be efficiently accessed by both worlds.

It is important to note that CMA should be used with caution, as reserving a large portion of memory for the CMA pool can impact overall system performance and memory utilization. Careful consideration of the trade-offs and system requirements is necessary when using CMA in a specific implementation.

Network

Question: Does OP-TEE support network communication in secure world?

Answer:

OP-TEE does not provide built-in support for direct network communication from within the secure world. The main reason for this is to maintain the security and isolation of the secure environment, as exposing networking capabilities directly to the secure world could introduce potential vulnerabilities or attack surfaces.

In most use cases, network communication is handled by normal world applications. If there is a need for secure communication involving trusted applications in the secure world, it is usually done indirectly:

1. Normal world applications handle network communication and exchange data with trusted applications in the secure world using the GlobalPlatform TEE Client API and shared memory.
2. Secure operations, such as encryption, decryption, or authentication, can be performed by trusted applications in the secure world. The normal world application can then send or receive the processed data over the network.
3. If end-to-end security is required, the normal world application can use additional encryption or authentication protocols, in conjunction with secure operations performed by trusted applications.

By handling network communication in the normal world and only exchanging data with the secure world as needed, OP-TEE maintains the isolation and security of the secure environment while still enabling secure network communication indirectly.

In summary, OP-TEE does not support direct network communication within the secure world. Instead, network communication is typically handled by normal world applications, which can securely exchange data with trusted applications in the secure world using the GlobalPlatform TEE Client API and shared memory.

Signature

Question: Is signature required when a normal world application request a service in secure world?

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.

Question: Does OP-TEE support to verify a signature of a TA in X.509 certificate?

Answer:

OP-TEE supports verifying the signature of a Trusted Application (TA) using X.509 certificates. This verification process is essential to ensure the authenticity and integrity of the TA before loading it into the secure world.

By default, OP-TEE uses a custom file format for TA binaries, which includes a signature generated with a private key. The signature is verified using the corresponding public key during the TA loading process. This public key is typically embedded into the OP-TEE core during the build process.

If you want to use X.509 certificates for verifying TA signatures, you can follow these steps:

1. Build the OP-TEE project with the `CFG_TA_ASN1_DER_FORMAT` configuration option enabled. This option tells OP-TEE to expect TAs to be signed with X.509 certificates instead of the default custom format.

```
make CFG_TA_ASN1_DER_FORMAT=y
```

2. Sign your TA using an X.509 certificate. You can use tools like OpenSSL to create a certificate and sign the TA binary.
3. Embed the root certificate (or a certificate chain) used to sign the TA into the OP-TEE core during the build process. This certificate is used to verify the TA signature at runtime.
4. When loading a TA, OP-TEE will use the embedded certificate(s) to verify the TA's signature in the X.509 certificate format.

Please note that you'll need to modify your build process and potentially some OP-TEE components to support this feature. Make sure to consult the OP-TEE documentation and source code for more information on how to enable and use X.509 certificates for TA signature verification.

Development

Question: Which languages do I use to develop apps in OP-TEE or to customize OP-TEE?

Answer:

To develop trusted applications for OP-TEE or to customize the OP-TEE framework, you primarily work with the C programming language. Both the OP-TEE OS (the secure kernel) and trusted applications are typically written in C to ensure low-level control, performance, and compatibility with the Arm TrustZone architecture.

1. Trusted Applications: To develop a trusted application (TA) for OP-TEE, you write your application code in C, using the TEE Internal Core API as specified by GlobalPlatform. This API provides functions and types for managing memory, cryptography, secure storage, and other operations within the trusted application.
2. Customizing OP-TEE: If you want to modify or extend the OP-TEE OS itself, you will also work with C. The OP-TEE OS codebase is written in C and can be customized to meet specific requirements or to add new features. However, customizing the OP-TEE OS requires a deeper understanding of the secure kernel's internals and the Arm TrustZone technology.

For normal world applications that communicate with the trusted applications, you can use various programming languages, as long as they can interface with the TEE Client API, which is usually provided as a C library. This means that you can use C, C++, or other languages that support C-compatible FFI (Foreign Function Interface) to interact with the secure world.

To summarize, you primarily use the C programming language to develop trusted applications and customize the OP-TEE framework, while normal world applications can be developed using various programming languages, as long as they can interface with the TEE Client API.

Question: Can I use other langauges?

Answer:

While the primary language for developing trusted applications (TAs) and customizing OP-TEE is C, it is possible to use other languages in some scenarios. However, this usually involves additional steps, and you may need to consider the limitations and compatibility of the chosen language with the TEE environment.

For normal world applications, you have more flexibility in choosing a programming language, as long as it can interface with the TEE Client API. Many popular programming languages have Foreign Function Interface (FFI) capabilities, which allow them to call C functions from native libraries. Some examples of such languages include:

- C++
- Python (using `ctypes` or `cffi`)
- Rust (using `bindgen` or `ctypes`)
- Go (using `cgo`)
- Node.js (using `ffi-napi` or `node-ffi`)

For trusted applications, using a language other than C is more challenging, as it must meet certain requirements, such as being compatible with the TEE environment, the TEE Internal Core API, and the constraints of the secure world. Some potential options include:

1. C++: You may be able to use a subset of C++ to develop trusted applications, as long as you handle low-level details and ensure compatibility with the TEE environment. However, you would need to be cautious about using certain C++ features, such as exceptions and the standard library, which may not be supported or might introduce security risks.
2. Rust: Rust is a systems programming language with strong safety guarantees, which could be suitable for developing trusted applications. There has been some work on using Rust for OP-TEE trusted applications, such as the `rust-optee-trustzone-sdk` project (https://github.com/apache/incubator-teaclave-trustzone-sdk). This project provides a Rust SDK for creating TAs and a set of bindings to the TEE Internal Core API.

Using other high-level languages like Python, Java, or JavaScript for trusted applications is generally not feasible due to their runtime environments, memory management, and performance characteristics, which may not be compatible with the constraints of the secure world.

In summary, while it is possible to use other languages for normal world applications and, to some extent, for trusted applications, the primary language for OP-TEE development remains C. When considering an alternative language, be sure to evaluate its compatibility with the TEE environment, the API, and the specific requirements of your application.

Question: Does OP-TEE support ARMv7 and ARMv8?

Answer:

Yes, OP-TEE supports both ARMv7 and ARMv8 architectures. OP-TEE is designed to work with Arm TrustZone technology, which is available on both ARMv7 and ARMv8-based processors. Arm TrustZone provides hardware-based isolation between the normal world and the secure world, allowing OP-TEE to implement a Trusted Execution Environment (TEE) that can run trusted applications securely.

While ARMv7 and ARMv8 have some differences in terms of instruction sets and features, OP-TEE is compatible with both architectures and can be used to build secure systems on devices using either ARMv7 or ARMv8 processors.

Question: Does OP-TEE have a shell?

Answer:

OP-TEE does not have a traditional shell like those found in Unix-like operating systems. However, OP-TEE provides a feature called the “Secure Shell” (S-Shell), which allows developers and users to interact with the secure world using a command-line interface. The S-Shell is primarily used for debugging purposes and is not intended for production use.

The S-Shell can be accessed using a UART (Universal Asynchronous Receiver/Transmitter) connection to the device running OP-TEE. This connection allows you to view log messages, interact with the secure world, and execute basic commands to manage trusted applications, memory, and other secure resources.

It is important to note that the S-Shell feature should be used carefully and only in controlled environments, as it could potentially expose sensitive information or allow unauthorized access to secure resources if misused.

In summary, while OP-TEE does not have a traditional shell, it does provide an S-Shell feature for debugging purposes. This feature should be used with caution, as it can potentially expose sensitive information or allow unauthorized access to secure resources if not properly managed.

Question: Can I create a TA at run-time?

Answer:

Creating a Trusted Application (TA) at runtime is not a typical use case for OP-TEE. TAs are usually developed, compiled, and signed offline before being deployed to the target device. In the normal process, TAs are loaded by the secure kernel (in the secure world) when requested by a client application (in the normal world).

However, if you have a specific use case that requires creating a TA at runtime, you could theoretically implement a custom mechanism for loading and running code in the secure world dynamically. This would involve significant modifications to the OP-TEE framework and would need to be done with extreme caution to ensure security is maintained.

Such a dynamic TA loading mechanism would likely require the following steps:

1. Develop and compile the TA code in a suitable format that can be loaded into the secure world at runtime.
2. Transfer the compiled TA code to the target device securely.
3. Modify the secure kernel to support loading and executing the TA code dynamically, including verifying its authenticity and integrity.
4. Implement proper access control and isolation mechanisms to ensure the newly loaded TA does not compromise the security of the existing TAs and the secure world.

It is crucial to emphasize that implementing a runtime TA creation mechanism is a complex and potentially risky endeavor, and it may introduce security vulnerabilities if not done correctly. You should carefully consider the security implications and whether the benefits of such a feature outweigh the risks before attempting to implement it.

Question: But what is TEECI_LoadSecureModule()

Answer:

My apologies for any confusion earlier. `TEECI_LoadSecureModule()` is not a part of the standard OP-TEE or GlobalPlatform TEE APIs. It seems to be a custom function name that might have been used in a specific implementation or context.

It’s important to reiterate that the standard OP-TEE and GlobalPlatform TEE APIs do not provide a built-in function for loading or creating Trusted Applications (TAs) at runtime. TAs are generally pre-built, signed, and deployed to the device as binaries. The secure kernel then loads the TA when requested by a client application.

If you have encountered `TEECI_LoadSecureModule()` in some source code or documentation, it might be a custom function that has been implemented for a specific use case or project. To understand its purpose and how it works, you would need to refer to the documentation or source code of that particular implementation.

Question: What are Static TA and Dynamic TA? Please also summarize them in a table.

Answer:

Static TA (Trusted Application) and Dynamic TA are two types of Trusted Applications used within the OP-TEE framework. Here's a summary of their differences in a table:

Feature	Static TA	Dynamic TA
Loading mechanism	Built into the OP-TEE core	Loaded at runtime as separate binaries
Memory management	Pre-allocated memory within the TEE core	Allocated dynamically when the TA is loaded
Lifespan	Lives for the entire duration of the TEE core	Loaded and unloaded on-demand, based on the client's request
Flexibility	Limited, as changes require recompiling the TEE core	Highly flexible, can be updated or replaced independently of the TEE core
Initialization process	Initializes during TEE core startup	Initializes when first invoked by a client
Performance	Faster, as no additional loading is needed	Slower, due to the need to load and initialize the TA at runtime
Storage	No separate storage required, as it's part of the TEE core	Requires storage for the separate TA binary

Static TAs are built directly into the OP-TEE core binary, which means they are always available when the TEE is running. They have pre-allocated memory within the TEE core and are initialized during the TEE core startup. This makes them faster in terms of performance but less flexible since any changes to a Static TA require recompiling the entire OP-TEE core.

Dynamic TAs, on the other hand, are loaded at runtime as separate binaries. They are more flexible and can be updated or replaced independently of the OP-TEE core. However, they may have a slightly slower performance due to the need to load and initialize the TA at runtime. Dynamic TAs are loaded and unloaded on-demand based on the client's request and have memory allocated dynamically when the TA is loaded.

Question: Does TEE define Static TA and Dynamic TA?

Answer:

The terms "Static TA" and "Dynamic TA" are specific to the OP-TEE implementation and are not part of the GlobalPlatform TEE (Trusted Execution Environment) specifications. The GlobalPlatform TEE specifications define the APIs and security requirements for trusted applications, but they do not dictate how TAs should be loaded, stored, or managed within a TEE implementation.

The concepts of static and dynamic TAs are unique to the OP-TEE project and serve as a way to distinguish between different loading and linking mechanisms for trusted applications within the OP-TEE framework. Other TEE implementations might use different terminology or approaches to manage trusted applications.

In summary, the terms "Static TA" and "Dynamic TA" are specific to OP-TEE and are not part of the GlobalPlatform TEE specifications.

Question: How to recover a crashed static TA?

Answer:

Recovering a crashed static TA in OP-TEE may not be a straightforward process, as static TAs are linked with the OP-TEE OS binary and run in the same context as the OS. However, you can try the following steps to recover from a crashed static TA:

1. Detect the crash: Set up a monitoring mechanism to detect when a static TA crashes or becomes unresponsive. This can be done by monitoring the return values of the TA's entry points, observing timeouts or abnormal behavior, and checking for signs of memory corruption.
2. Terminate the TA session: In case of a detected crash, terminate the session with the affected static TA. Inform the client application (CA) in the normal world about the issue, and potentially return an error code to signify the crash.
3. Clean up resources: Release any resources held by the crashed static TA, such as memory allocations, file handles, or cryptographic keys.
4. Reinitialize the TA: As static TAs are linked with the OP-TEE OS binary, the only way to reinitialize the TA is by resetting the entire secure world. This can be done by rebooting the system or triggering a system reset. Note that this approach may also affect other TAs and the OP-TEE OS itself, so it should be used with caution.
5. Restore the state (optional): If the static TA maintains a persistent state in secure storage, you might need to restore the state after reinitialization. This can be done by reading the state from secure storage and passing it back to the TA when it starts.
6. Resume operation: Once the static TA is reinitialized and its state is restored (if applicable), you can resume the normal operation by relaunching the client application and re-establishing the session with the TA.

Please note that the recovery process for a crashed static TA is not ideal, as it may involve resetting the entire secure world. It is essential to design and implement the static TA with robust error handling and recovery mechanisms to minimize the risk of crashes. Additionally, you might consider using dynamic TAs instead of static TAs, as dynamic TAs offer better isolation and can be loaded and unloaded independently, making recovery easier in case of crashes.

Question: How to recover a crashed dynamic TA?

Answer:

Recovering a crashed dynamic TA in OP-TEE is generally more manageable than recovering a static TA since dynamic TAs run in separate contexts and can be loaded and unloaded independently. Here's a suggested process to recover a crashed dynamic TA:

1. Detect the crash: Set up a monitoring mechanism to detect when a dynamic TA crashes or becomes unresponsive. This can be done by monitoring the return values of the TA's entry points, observing timeouts, or abnormal behavior.
2. Terminate the TA session: In case of a detected crash, terminate the session with the affected dynamic TA. Inform the client application (CA) in the normal world about the issue, and potentially return an error code to signify the crash.
3. Clean up resources: Release any resources held by the crashed dynamic TA, such as memory allocations, file handles, or cryptographic keys.
4. Unload the TA: Unload the crashed dynamic TA from the OP-TEE OS by closing the session associated with the TA. This should release the resources held by the TA and remove it from memory.
5. Reload the TA: Load a new instance of the dynamic TA using the same TA UUID. This can be done by requesting a new session with the TA from the client application.
6. Restore the state (optional): If the dynamic TA maintains a persistent state in secure storage, you might need to restore the state after reloading the TA. This can be done by reading the state from secure storage and passing it back to the TA when it starts.
7. Resume operation: Once the dynamic TA is reloaded and its state is restored (if applicable), you can resume normal operation by relaunching the client application and re-establishing the session with the TA.

Please note that it’s essential to design and implement the dynamic TA with robust error handling and recovery mechanisms to minimize the risk of crashes. Be sure to thoroughly test and validate the TA to ensure its stability and reliability.

Qualcomm

Question: Does Qualcomm support OP-TEE?

Answer:

As of my knowledge cutoff in September 2021, Qualcomm does not officially support OP-TEE for their platforms. Instead, they provide their own Trusted Execution Environment (TEE) solution, known as Qualcomm Trusted Execution Environment (QTEE). QTEE is based on ARM TrustZone technology and is designed to provide a secure environment for running trusted applications on Qualcomm Snapdragon processors.

QTEE offers similar functionality to OP-TEE, providing a secure environment for running sensitive code and protecting sensitive data from unauthorized access or tampering. It includes features such as secure boot, secure storage, cryptographic services, and key management.

While Qualcomm officially supports QTEE on their platforms, it is possible that some developers or third-party companies have attempted to port OP-TEE to Qualcomm devices. However, such ports may not be officially supported by Qualcomm and may not offer the same level of integration, performance, or security as the QTEE solution.

Question: Is QTEE open source?

Answer:

Qualcomm Trusted Execution Environment (QTEE) is not fully open-source. While some components and tools related to QTEE are available as open-source projects, the core implementation of QTEE is proprietary and closed-source.

Qualcomm provides certain tools, libraries, and APIs to facilitate the development of trusted applications (TAs) for QTEE, and some of these resources may be available as open-source projects. However, the QTEE OS and other critical components are closed-source and maintained by Qualcomm.

This is in contrast to OP-TEE, which is an open-source TEE solution maintained by Linaro and supported by various industry partners. With OP-TEE, developers have access to the full source code, enabling them to examine, modify, and contribute to the project as needed.

DRM

Question: How do I implement DRM of content decryption with OP-TEE?

Answer:

Implementing DRM content decryption with OP-TEE requires the development of a Trusted Application (TA) that handles the decryption process securely within the TEE environment. Here is a step-by-step guide to help you implement DRM content decryption using OP-TEE:

1. Define decryption requirements: Identify the specific encryption schemes and formats used for the protected content, such as AES-CTR, AES-CBC, or others.

2. Develop a Trusted Application: Create a TA that handles the decryption process. The TA should receive encrypted content and any necessary decryption keys or metadata from the normal world and perform the decryption securely within the TEE environment. It should also return the decrypted content to the normal world application for playback.

```
TEE_Result TA_InvokeCommandEntryPoint(void *session_context, uint32_t command_id,
                                      uint32_t param_types, TEE_Param params[TEE_NUM_PARAMS]) {
    switch (command_id) {
        case TA_DRM_DECRYPT_CMD:
            return ta_drm_decrypt(param_types, params);
        ...
    }
}
```

3. Implement decryption functionality: Within the TA, implement the necessary decryption algorithms and functionality. You can use the cryptographic operations provided by the TEE Internal API to perform the decryption.

```
static TEE_Result ta_drm_decrypt(uint32_t param_types, TEE_Param params[TEE_NUM_PARAMS]) {
    TEE_Result res;
    TEE_OperationHandle op = TEE_HANDLE_NULL;
    TEE_ObjectHandle key_handle = TEE_HANDLE_NULL;
    ...
    // Create a transient key object and set the decryption key
    res = TEE_AllocateTransientObject(TEE_TYPE_AES, key_size, &key_handle);
    res = TEE_PopulateTransientObject(key_handle, key_attr, 1);

    // Allocate a new operation and set the key for decryption
    res = TEE_AllocateOperation(&op, TEE_ALG_AES_CBC_NOPAD, TEE_MODE_DECRYPT, key_size);
    res = TEE_SetOperationKey(op, key_handle);

    // Perform the decryption
    res = TEE_CipherDoFinal(op, encrypted_data, encrypted_data_size, decrypted_data, &decrypted_data_size
    ...
}
```

4. Integrate with the normal world application: Develop a normal world application that communicates with the TA using OP-TEE's GlobalPlatform TEE Client API. The normal world application should be responsible for obtaining encrypted content, the decryption key, and any necessary metadata from the DRM system, and then passing this information to the TA for decryption.

```
TEEC_Result res;
TEEC_Session session;
TEEC_Operation op;
...
// Prepare the operation parameters
op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INPUT, TEEC_MEMREF_TEMP_INPUT, TEEC_MEMREF_TEMP_OUTPUT
op.params[0].tmpref.buffer = encrypted_data;
op.params[0].tmpref.size = encrypted_data_size;
op.params[1].tmpref.buffer = decryption_key;
op.params[1].tmpref.size = decryption_key_size;
op.params[2].tmpref.buffer = decrypted_data;
op.params[2].tmpref.size = decrypted_data_size;

// Invoke the TA command for decryption
res = TEEC_InvokeCommand(&session, TA_DRM_DECRYPT_CMD, &op, NULL);
```

5. Test and validate: Test your TA and normal world application to ensure proper decryption and playback of the protected content. Verify that the content is decrypted securely within the TEE environment and is protected from unauthorized access or tampering.

By following these steps, you can implement DRM content decryption with OP-TEE, ensuring that the decryption process is secure and protected within

---

Question: So the content is decrypted in TA in secure world. Right?

Answer:

Yes, that’s correct. In the described DRM implementation, the content is decrypted within the Trusted Application (TA) running in the secure world of OP-TEE. The TA receives encrypted content and decryption keys from the normal world application and performs the decryption securely within the TEE environment. The decrypted content is then returned to the normal world application for playback. This approach ensures that sensitive information, such as decryption keys and decrypted content, is protected within the secure world, preventing unauthorized access or tampering.

Question: But does it slow down the playback speed because of switching between normal world and secure world?

Answer:

Switching between the normal world and the secure world does introduce some overhead, as the system needs to perform context switching and maintain the isolation between the two environments. However, this overhead is generally small compared to the overall processing time required for content decryption and playback.

The exact performance impact will depend on various factors, such as the hardware platform, the complexity of the decryption algorithm, and the size of the content being processed. In most cases, the benefits of secure content decryption within the TEE outweigh the slight performance overhead introduced by the world-switching.

To minimize the performance impact, you can optimize the design of your Trusted Application and normal world application. For example, you can:

1. Minimize the number of world switches by processing larger chunks of encrypted content in a single invocation of the Trusted Application.
2. Optimize the cryptographic operations within the Trusted Application by leveraging hardware-accelerated cryptographic functions, if available on your platform.
3. Optimize data handling and buffering between the normal world application and the Trusted Application to reduce data copying and memory allocation overhead.

By carefully designing and optimizing your implementation, you can ensure that the performance impact of secure content decryption in the TEE is minimized while still providing robust security for your DRM system.

Question: Can I use multi cores to solve the problem? For example, one core decrypts content and another core play content at the same time.

Answer:

Yes, using multi-core processing can help improve the performance of your DRM implementation by parallelizing the decryption and playback processes. If your platform supports multiple cores, you can assign different tasks to different cores to minimize the overhead introduced by secure world and normal world switching. Here’s a general approach to achieve this:

1. Assign content decryption to one core: Designate one core to handle the decryption process within the Trusted Application in the secure world. This core will be responsible for processing encrypted content and performing the decryption using the provided keys.

2. Assign content playback to another core: Designate a different core to handle the playback process in the normal world. This core will be responsible for receiving the decrypted content from the Trusted Application and managing the playback process, including rendering, audio synchronization, and user interaction.

3. Implement inter-core communication: Develop a mechanism for inter-core communication to allow the decryption core and the playback core to exchange data efficiently. This can be achieved using shared memory buffers, message queues, or other communication techniques suitable for your platform.

4. Manage synchronization and buffering: Implement synchronization and buffering mechanisms to ensure smooth playback while maintaining the parallelism between decryption and playback processes. This may involve using semaphores, mutexes, or other synchronization primitives to coordinate data exchange between the cores and prevent buffer overflows or underflows.

By using multi-core processing, you can offload the decryption process to a separate core, reducing the impact of secure world and normal world switching on playback performance. However, keep in mind that implementing a multi-core DRM solution can be more complex, as it requires careful design and synchronization to ensure smooth operation and proper security. Be sure to thoroughly test and validate your implementation to ensure it meets the performance and security requirements of your DRM system.

Question: Does OP-TEE support TUI?

Answer:

Yes, OP-TEE supports Trusted User Interface (TUI), which allows developers to create secure and isolated user interfaces for Trusted Applications (TAs) running within the TEE environment. TUI ensures that sensitive user inputs, such as passwords, PINs, or biometric data, are protected from potential threats in the normal world by providing a secure, isolated display and input handling mechanism.

OP-TEE’s TUI implementation leverages ARM TrustZone technology to protect the display and input resources from unauthorized access or tampering. The TUI is rendered and managed within the secure world, and the input data collected through the TUI is kept secure, preventing any leakage to the normal world.

To implement a TUI in your OP-TEE project, you’ll need to follow these steps:

1. Develop a Trusted Application: Create a TA that handles the secure processing required by your application, including any input collected through the TUI.
2. Implement the TUI: Design and develop the TUI within your TA, using the TEE Internal API and any available TUI-specific extensions provided by your platform or device vendor. This may involve creating secure framebuffer objects, managing touch input events, and handling user interaction.
3. Integrate with the normal world application: Develop a normal world application that communicates with your TA, triggering the display of the TUI and receiving any results produced by the TA after processing the secure input.
4. Test and validate: Test your TA, TUI, and normal world application to ensure proper functionality, security, and user experience.

Please note that the specifics of implementing a TUI may vary depending on the hardware platform and device vendor, as different vendors may provide their own TUI extensions or libraries for use with OP-TEE. Consult the documentation and resources provided by your platform or device vendor for detailed guidance on implementing a TUI in your particular environment.

Question: Can I develop a TA with TUI to play video in OP-TEE?

Answer:

Developing a Trusted Application (TA) with a Trusted User Interface (TUI) to play video within the OP-TEE secure world is theoretically possible, but there are several practical challenges and limitations to consider. Here are some of the primary concerns:

1. Performance: Video playback typically requires significant processing power, and the secure world in OP-TEE is generally not designed for high-performance tasks. Video playback within a TA might result in suboptimal performance compared to playback in the normal world.
2. Hardware support: Video playback often relies on hardware acceleration (e.g., GPU or video decoding hardware) to provide smooth and efficient performance. Access to such hardware components from the secure world might not be available or may require platform-specific modifications.
3. Limited resources: OP-TEE and TAs typically have access to a limited amount of memory and processing resources compared to the normal world. This constraint might impact video playback quality and performance.
4. Complexity: Implementing video playback within a TA requires handling various codecs, formats, and hardware components within the secure environment, which can significantly increase the complexity of your solution.
5. Use case justification: Playing video in a TA with TUI may not be necessary in most use cases. It’s essential to evaluate if the benefits of secure video playback within the TEE outweigh the complexity, performance, and resource implications.

If you require a secure video playback solution, consider alternative approaches, such as using a DRM system in the normal world and leveraging the TEE for securely handling decryption keys and content decryption. This way, you can still ensure the security of sensitive information while benefiting from the performance and hardware capabilities available in the normal world.

If you have a specific use case that mandates secure video playback within the TEE, consult your hardware platform and device vendor documentation to understand the capabilities and limitations of your target environment. You may need to work closely with the vendor to develop a custom solution that meets your requirements while addressing the challenges and limitations mentioned above.

Question: How do I implement a DRM solution using ClearKey in OP-TEE?

Answer:

To implement a DRM solution using ClearKey in OP-TEE, you'll need to develop a Trusted Application (TA) that handles the decryption process within the secure world of OP-TEE, and a normal world application (e.g., a browser or media player) that communicates with the TA for decrypting media content. Here are the general steps:

1. Develop a Trusted Application (TA): Create a TA that securely handles decryption keys and content decryption within the TEE environment. The TA will receive encrypted content and decryption keys from the normal world application and perform the decryption securely within the TEE.

2. Implement the ClearKey decryption algorithm: Since ClearKey is a simple and open-source key system, you can readily find implementations or libraries that can be integrated into your TA. You will need to implement the decryption algorithm (usually AES-128-CBC or AES-128-CTR) within the TA to decrypt the content.

3. Develop a normal world application: Create a browser or media player application that uses the Encrypted Media Extensions (EME) API with the ClearKey key system. This application will be responsible for handling media playback, parsing the encrypted content, and extracting the necessary metadata (e.g., content encryption keys, initialization vectors) to pass to the TA for decryption.

4. Communication between normal world application and TA: Establish a secure communication channel between your normal world application and the TA. Your normal world application will send encrypted media content, keys, and other required metadata to the TA, which will then decrypt the content and return it to the normal world application for playback.

5. Test and validate your implementation: Ensure that your DRM solution works correctly by testing it with various encrypted media content and key configurations. Make sure the implementation provides the necessary security features to protect the content and keys from unauthorized access or tampering.

Keep in mind that ClearKey is a basic key system primarily intended for testing and development purposes. In a production environment, you may want to use a more robust and secure DRM system, such as Widevine or PlayReady, depending on your requirements and platform support. Additionally, be aware that implementing a DRM solution in OP-TEE can be complex, and you'll need to consider the hardware and platform limitations, performance, and security requirements for your specific use case.

Secure Storage

Question: Does OP-TEE support file system?

Answer:

OP-TEE itself does not provide a full-fledged file system like those found in traditional operating systems. However, it does offer secure storage functionality that allows trusted applications to store and manage persistent data securely. The secure storage feature in OP-TEE is designed to protect sensitive data from unauthorized access and tampering.

The secure storage is implemented using a key-value store model, where each object (file) is associated with a unique identifier. Trusted applications can use the TEE Internal Core API to read, write, and delete objects in the secure storage. This API provides functions like `TEE_CreatePersistentObject`, `TEE_OpenPersistentObject`, `TEE_ReadObjectData`, `TEE_WriteObjectData`, and `TEE_CloseAndDeletePersistentObject`.

Under the hood, the secure storage in OP-TEE uses encryption and authentication techniques to protect the stored data. The actual storage is typically located in the normal world file system, but the data is encrypted and protected in such a way that it can only be accessed and manipulated by the intended trusted applications in the secure world.

In summary, OP-TEE does not have a traditional file system, but it does provide secure storage functionality for trusted applications to store and manage sensitive data securely.

Question: How does OP-TEE generate SSK from HUK in tee_fs_init_key_manager()?

Answer:

In OP-TEE, the `tee_fs_init_key_manager()` function is responsible for generating the Secure Storage Key (SSK) from the Hardware Unique Key (HUK). This process takes place during the initialization of the secure file system. The function can be found in the `core/tee/tee_fs_key_manager.c` file in the OP-TEE source code.

Here's an overview of the steps involved in generating the SSK from the HUK in `tee_fs_init_key_manager()`:

1. Retrieve the HUK: The HUK is obtained by calling `tee_otp_get_hw_unique_key()`. This function retrieves the device-specific hardware unique key and stores it in a `tee_hw_unique_key` structure.

```
struct tee_hw_unique_key huk = {0};
if (tee_otp_get_hw_unique_key(&huk) != TEE_SUCCESS)
    return TEE_ERROR_SECURITY;
```

2. Create a message containing a static string and the device's chip ID: The message is created by concatenating a static string (e.g., "ONLY_FOR_tee_fs_ssk") and the device's chip ID, which can be obtained using `tee_otp_get_die_id()`.

```
uint8_t message[sizeof(ssk_gen_message) + TEE_FS_KM_CHIP_ID_LENGTH] = {0};
memcpy(message, ssk_gen_message, sizeof(ssk_gen_message));
if (tee_otp_get_die_id(message + sizeof(ssk_gen_message), TEE_FS_KM_CHIP_ID_LENGTH) !=
    TEE_SUCCESS)
    return TEE_ERROR_SECURITY;
```

3. Derive the SSK: The SSK is derived from the HUK using the HMAC-SHA-256 function. The HMAC function takes the HUK as the key and the message created in the previous step as input data. The result is a 256-bit output, which is used as the SSK.

```
uint8_t digest[TEE_FS_KM_HMAC_SHA256_DIGEST_SIZE] = {0};
TEE_Result res = tee_do_hmac_sha256(huk.data, sizeof(huk.data), message,
                                    sizeof(message), digest);
if (res != TEE_SUCCESS)
    return res;

memcpy(tee_fs_ssk.key, digest, sizeof(tee_fs_ssk.key));
```

After these steps, the SSK is generated and stored in the `tee_fs_ssk` global variable, which is used by the secure file system for encryption and decryption.

In summary, the `tee_fs_init_key_manager()` function in OP-TEE generates the SSK from the HUK by first retrieving the HUK and then using the HMAC-SHA-256 function with a message consisting of a static string and the device's chip ID. The derived SSK is then used for encryption and decryption in the secure file system.

Question: In the function, can I read the content of the HUK?

Answer:

In OP-TEE's `tee_fs_init_key_manager()` function, the Hardware Unique Key (HUK) is retrieved and stored in a local variable. While it is technically possible to read the content of the HUK within this function, doing so would pose a security risk. The HUK is intended to be a secret key, unique to each device, and should not be exposed to unauthorized parties.

If you want to inspect the HUK for debugging or development purposes, you can use debug logging functions (such as `DMSG()` or `EMSG()`) to print the content of the HUK within the `tee_fs_init_key_manager()` function. However, you should remove or disable such logging statements before deploying your OP-TEE build in a production environment, to ensure the security of the HUK.

Keep in mind that exposing the HUK can lead to potential security vulnerabilities, as it serves as the basis for deriving various other keys (such as the Secure Storage Key) used in OP-TEE's secure services. It is crucial to handle the HUK and other sensitive data carefully to maintain the overall security of the system.

Question: How do I protect HUK from reading in OP-TEE?

Answer:

Protecting the Hardware Unique Key (HUK) is essential for ensuring the security of an OP-TEE-based system. Since the HUK is used as the basis for deriving other keys and securing various sensitive operations, it is crucial to keep it confidential. Here are some best practices for protecting the HUK in OP-TEE:

1. Limit access to the HUK: Ensure that the HUK is only accessed by the necessary parts of the OP-TEE core and limit its exposure to other components. This can be achieved by using the HUK only within trusted functions and avoiding passing it as a parameter to other functions or storing it in global variables.

2. Avoid logging the HUK: When developing or debugging OP-TEE, avoid logging the HUK or any derived keys. If you need to debug HUK-related issues, make sure to remove or disable any logging statements before deploying your build in a production environment.

3. Protect HUK storage in hardware: The HUK is usually stored in a secure, tamper-resistant location within the device hardware, such as a one-time programmable (OTP) memory or a secure element. Ensure that the hardware implementation adequately protects the HUK from unauthorized access or tampering.

4. Protect the HUK during retrieval: When the HUK is retrieved from hardware, use platform-specific secure functions to access it, and make sure that the retrieval process is implemented securely to prevent unauthorized access to the HUK.

5. Secure software implementation: Follow secure coding practices to minimize the risk of software vulnerabilities that could lead to unauthorized access to the HUK or other sensitive data. Regularly review and update the OP-TEE implementation to incorporate the latest security patches and enhancements.

6. Secure boot process: Implement a secure boot process to ensure that only authorized and trusted code can execute on the device. This will help prevent potential attacks that could compromise the HUK or other sensitive information.

By following these best practices, you can help protect the HUK from unauthorized access and maintain the security of your OP-TEE-based system. Remember that safeguarding the HUK is crucial, as it serves as the foundation for various other keys and secure operations within the system.

Question: Is SSK defined in GlobalPlatform?

Answer:

The Secure Storage Key (SSK) concept, as implemented in OP-TEE, is not directly defined by the GlobalPlatform TEE specifications. The GlobalPlatform TEE specifications define a set of APIs and security requirements for Trusted Applications and the TEE itself but do not specifically address the implementation details of secure storage.

In OP-TEE, the SSK is used for encrypting and decrypting data stored in the secure storage. It is derived from the Hardware Unique Key (HUK) using a Key Derivation Function (KDF). While this implementation choice is specific to OP-TEE, it aligns with the broader security goals outlined by the GlobalPlatform TEE specifications, such as ensuring the confidentiality and integrity of data in secure storage.

The GlobalPlatform TEE specifications provide guidance on general security requirements, but individual TEE implementations, such as OP-TEE, can choose how to implement specific features like secure storage and key management, as long as they meet the overall security objectives of the specifications.

Recover OP-TEE

Question: How to detect OP-TEE dead?

Answer:

Detecting if OP-TEE is dead or non-responsive can be done through various mechanisms. Here are some approaches:

1. Timeouts: Set a timeout for the normal world when waiting for a response from OP-TEE. If the timeout expires and no response has been received, consider OP-TEE non-responsive or dead.
2. Heartbeat mechanism: Implement a periodic “heartbeat” or “ping” mechanism between the normal world and OP-TEE. If the normal world does not receive a response within a predefined time interval, consider OP-TEE non-responsive or dead.
3. Error checking: Monitor the normal world’s interactions with OP-TEE for unexpected errors or responses. If such errors occur, it may indicate a problem with OP-TEE’s operation.
4. System logs: Analyze system logs and debug messages to identify any unexpected behaviors or error messages from OP-TEE.
5. External monitoring: Use an external hardware or software monitoring solution to detect any anomalies in the operation of the device or the secure world.

In case OP-TEE is determined to be non-responsive or dead, a proper recovery or debugging mechanism should be in place to handle such situations. This might include resetting the device, collecting debug information, or triggering a fail-safe mechanism.

Question: How to automatically recovery the crashed OP-TEE?

Answer:

Automatically recovering a crashed OP-TEE can be challenging, as the recovery process depends on the root cause of the crash and the available recovery mechanisms. However, you can implement a watchdog mechanism to detect and attempt to recover from a crash automatically. Here are some general steps to consider:

1. Implement a watchdog timer: Utilize a hardware or software watchdog timer to monitor OP-TEE’s operation. The timer should be periodically reset by the OP-TEE under normal operation. If OP-TEE crashes, the timer will not be reset, and the watchdog mechanism will trigger a recovery action after a predefined timeout.
2. Monitor normal world and secure world communication: Regularly check the communication between the normal world and the secure world. If the communication stalls or encounters unexpected errors, trigger a recovery action.
3. Define recovery actions: Design recovery actions to take when a crash is detected. Possible actions include: a. Reset the device. b. Reset the secure world or restart OP-TEE without affecting the normal world. c. Switch to a failsafe mechanism or an alternative trusted execution environment.
4. Test and validate: Thoroughly test and validate the automatic recovery mechanism to ensure it works as intended and does not introduce additional security risks or undesired side effects.
5. Implement error reporting: After a crash and recovery, collect and report relevant logs, error messages, or diagnostic information to aid in future debugging and analysis.

Please note that the specific implementation of an automatic recovery mechanism will depend on your device, its hardware and software stack, and the nature of the crash. Implementing such a mechanism can be complex and requires careful consideration of security implications and system behavior. It is essential to have a well-defined recovery plan and debugging mechanism in place for such situations.

Question: Can I develop a dual OP-TEE so that if one is crashed, run another.

Answer:

Developing a dual OP-TEE setup to provide redundancy in case one crashes is an interesting idea. Although it might not be straightforward, you can consider the following approach:

1. Run two instances of OP-TEE: Modify the bootloader and memory layout to accommodate two separate OP-TEE instances running on different cores or partitions of the device. Each OP-TEE instance should have its own secure memory space, and they should not interfere with each other.

2. Synchronize states: Implement a mechanism to synchronize the state and configuration between the two OP-TEE instances. This will ensure that if one instance crashes, the other can take over seamlessly.
3. Monitor OP-TEE health: Develop a health monitoring system that continuously checks the status of both OP-TEE instances. If an issue is detected with one instance, the monitoring system should automatically switch to the other instance.
4. Coordinate normal world communication: Modify the normal world OS and the secure world communication layer to support communicating with multiple OP-TEE instances. This may involve load balancing and failover mechanisms to distribute requests and handle failures.
5. Handle failover: Implement a failover mechanism that quickly switches between OP-TEE instances in case of a crash. This could include redirecting pending requests to the other instance, transferring the state, and ensuring secure storage continuity.

Please note that implementing a dual OP-TEE setup can be complex and may introduce additional security risks and performance overhead. It is essential to carefully analyze the system design, security implications, and potential trade-offs before implementing such a solution. Additionally, thorough testing and validation will be required to ensure the dual OP-TEE setup functions as intended without introducing new vulnerabilities or unexpected side effects.

Question: How to detect if OP-TEE is tamperred at run-time?

Answer:

Detecting if OP-TEE has been tampered with at runtime is a challenging task, as an attacker who successfully infiltrates the TEE environment could attempt to hide their activities. However, you can implement some techniques to help identify potential tampering:

1. Runtime integrity checks: Perform periodic integrity checks of the OP-TEE code and data segments while the system is running. This can be achieved by calculating a hash or checksum of these memory regions and comparing them with their expected values.
2. Hardware-based security features: Leverage hardware security features like ARM TrustZone's secure memory, which can prevent unauthorized access to sensitive memory regions. Secure memory can protect critical OP-TEE data structures and code from being tampered with by external entities.
3. Secure boot: Ensure that the system uses secure boot, which verifies the integrity of the bootloader, kernel, and OP-TEE components during the boot process. A secure boot chain can minimize the chances of tampering with OP-TEE before it starts running.
4. Hardware security modules (HSMs): HSMs can provide additional protection against tampering by securely storing cryptographic keys and performing sensitive operations within the secure hardware environment.
5. Monitor system behavior: Keep an eye on the system's behavior and resource utilization. Unusual patterns or unexpected resource usage may be indicative of tampering or an ongoing attack.
6. Logging and auditing: Maintain detailed logs of activities within the OP-TEE environment and perform regular audits to identify potential anomalies or unauthorized actions.
7. Use of external monitoring tools: Employ external runtime monitoring tools that can analyze the runtime behavior of the system to detect anomalies or signs of tampering.

Keep in mind that no single technique can provide complete protection against tampering. It is crucial to use a combination of methods and maintain a strong security posture through continuous monitoring, testing, and updating of your system's security mechanisms.

Attack

Question: Whare are kinds of methods to attack OP-TEE?

Answer:

There are various methods an attacker might use to target OP-TEE. Some of the common attack vectors include:

1. Exploiting software vulnerabilities: Attackers can exploit bugs or vulnerabilities in the OP-TEE code, such as buffer overflows, use-after-free, or format string vulnerabilities, to gain unauthorized access or escalate privileges within the TEE.
2. Side-channel attacks: These attacks focus on extracting sensitive information by observing the physical characteristics of the system, such as power consumption, electromagnetic emissions, or timing information. Examples include cache-timing attacks, power analysis attacks, and differential fault analysis.
3. Reverse engineering: Attackers can reverse-engineer the OP-TEE firmware to discover vulnerabilities, understand internal structures, or extract sensitive information such as cryptographic keys.
4. Bypassing secure boot: If an attacker can compromise the secure boot process, they can potentially tamper with or replace the OP-TEE components before they are loaded into memory, undermining the entire TEE's security.
5. Physical attacks: Sophisticated attackers may attempt to directly manipulate the hardware, for example, by probing the memory bus or tampering with the hardware security modules (HSMs).
6. Social engineering: Attackers can use social engineering techniques to trick developers or users into disclosing sensitive information or installing malicious software that compromises the TEE environment.
7. Compromising the rich execution environment (REE): If an attacker gains control over the REE, they might attempt to exploit the communication channel between the REE and the TEE to manipulate or intercept sensitive data.
8. Fault injection attacks: By inducing faults in the hardware, an attacker can cause the system to behave unexpectedly, potentially revealing sensitive information or bypassing security mechanisms. Examples include voltage glitching, clock glitching, and laser fault injection.
9. Rollback attacks: An attacker might attempt to revert the TEE to a previous, vulnerable state, allowing them to exploit known vulnerabilities or bypass security patches.

To mitigate these risks, it's essential to maintain a strong security posture, implement secure coding practices, and regularly update OP-TEE to the latest version. Additionally, employ hardware-based security features and perform regular security audits to identify potential vulnerabilities.