

# Introduction: Building a Foundation for Secure Production Systems

Moving from prototypes to production demands a professional toolchain. For the NXP i.MX 9 series, this means leveraging three core technologies:

- 1. **NXP i.MX 9 Processors:** These SoCs are designed for industrial and IoT applications, featuring the **EdgeLock® Secure Enclave**, a physically isolated security subsystem with its own CPU and memory. This provides a hardware root of trust and an extra layer of protection beyond standard TEEs, making it an ideal foundation for secure products. NXP further supports this with a 15-year product longevity program, ensuring a stable supply for long-lifecycle devices.
- 2. **The Yocto Project:** This is the industry-standard framework for creating custom, production-ready embedded Linux distributions.<sup>1</sup> It offers unparalleled control over every software component in your system, ensuring your final product is lean, stable, and precisely tailored to your needs.
- 3. **Docker:** By encapsulating the entire Yocto build environment within a Docker container, we eliminate the "works on my machine" problem. This ensures that every developer on your team, as well as your continuous integration (CI) pipeline, uses an identical, reproducible build environment, dramatically increasing efficiency and reducing errors.<sup>3</sup>

This guide will walk you through the complete end-to-end process, from setting up the environment to deploying and running your own custom Rust-based Trusted Application on an i.MX 93 evaluation kit.

---

## Chapter 1: Assembling Your Professional Toolkit

Before we begin, ensure you have the necessary hardware and a properly configured host machine.

### 1.1. Hardware Requirements

Item	Specification	Purpose & Notes
------	---------------	-----------------

<b>Evaluation Board</b>	NXP i.MX 93 EVK	The official development platform for the i.MX 93 SoC. It provides all necessary interfaces for development and debugging. (Approx. \$622-\$635 USD)
<b>Storage</b>	MicroSD Card (32GB+, Class 10/UHS-I)	To store the bootloader and the complete Linux OS image generated by Yocto.
<b>Power Supply</b>	USB-C Power Adapter (5V, 3A)	To provide stable power to the EVK.
<b>Cables</b>	2x USB-C to USB-A Cables	One for the debug serial console and one for flashing via the UUU tool.

## 1.2. Host System Requirements

- **Operating System:** A 64-bit Linux distribution. **Ubuntu 22.04 LTS** is strongly recommended as it is the primary validation platform for both Yocto and NXP's BSP.<sup>5</sup>
- **Software:** Docker Engine. Follow the official installation guide at [docs.docker.com](https://docs.docker.com) to install Docker on your host machine.<sup>7</sup> Ensure you can run Docker commands without sudo by adding your user to the docker group.<sup>4</sup>
- **Disk Space:** A minimum of 250 GB of free disk space is recommended for the Yocto build process, which includes source code, toolchains, and build artifacts.<sup>8</sup>

---

## Chapter 2: The Dockerized Yocto Build Environment

Our first step is to create a self-contained, portable build environment using Docker. This is based on NXP's official imx-docker project, a community best practice.<sup>4</sup>

### 2.1. Create the Dockerfile

Create a file named Dockerfile with the following content. This file instructs Docker to build an image based on Ubuntu 22.04 and install all the packages required by the

Yocto Project.<sup>6</sup>

Dockerfile

```
# Use Ubuntu 22.04 as the base image
```

```
FROM ubuntu:22.04
```

```
# Avoid interactive prompts during package installation
```

```
ARG DEBIAN_FRONTEND=noninteractive
```

```
# Update package lists and install all necessary dependencies for Yocto
```

```
RUN apt-get update && apt-get upgrade -y && \
```

```
apt-get install -y gawk wget git diffstat unzip texinfo gcc-multilib \
```

```
build-essential chrpath socat cpio python3 python3-pip python3-pexpect \
```

```
xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa \
```

```
libssl1.2-dev pylint3 xterm python3-subunit mesa-common-dev zstd liblz4-tool \
```

```
file locales sudo && \
```

```
locale-gen en_US.UTF-8
```

```
# Install the 'repo' tool for managing Yocto sources
```

```
RUN curl https://storage.googleapis.com/git-repo-downloads/repo > /usr/local/bin/repo && \
```

```
chmod a+rx /usr/local/bin/repo
```

```
# Set up a non-root user for the build process
```

```
RUN useradd -ms /bin/bash yoctouser && \
```

```
echo "yoctouser ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
```

```
USER yoctouser
```

```
WORKDIR /home/yoctouser
```

```
# Set the locale
```

```
ENV LC_ALL en_US.UTF-8
```

```
ENV LANG en_US.UTF-8
```

```
ENV LANGUAGE en_US.UTF-8
```

## 2.2. Build and Run the Container

1. **Build the Docker Image:** Open a terminal in the same directory as your Dockerfile and run:

```
Bash
```

```
docker build -t imx-yocto-builder.
```

2. **Create a Workspace:** On your host machine, create a directory where all your Yocto project files will be stored. This directory will be mounted into the container to ensure your work persists.

Bash

```
mkdir ~/imx-yocto-workspace
```

3. **Launch the Container:** Start an interactive session inside your new container.

Bash

```
docker run -it --rm \
-v ~/imx-yocto-workspace:/home/yoctouser/workspace \
imx-yocto-builder /bin/bash
```

You are now inside the container, at the /home/yoctouser prompt. Your host's ~/imx-yocto-workspace directory is available at ~/workspace.

---

## Chapter 3: Building the NXP BSP with OP-TEE Enabled

Now, inside the container, we will download NXP's Board Support Package (BSP) and build a base Linux image with OP-TEE support.

### 3.1. Fetch the Yocto Sources

NXP uses the repo tool to manage the multiple Git repositories that make up its BSP.

Bash

```
# Navigate to your persistent workspace
```

```
cd ~/workspace
```

```
# Configure Git with your identity
```

```
git config --global user.name "Your Name"
```

```
git config --global user.email "you@example.com"
```

```
# Initialize the repo manifest for a specific i.MX release
```

```
# We use a recent, stable release for i.MX 9 platforms
```

```
repo init -u https://github.com/nxp-imx/imx-manifest -b imx-linux-scarthgap -m
```

imx-6.6.52-2.2.0.xml

```
# Synchronize all the source code repositories (this will take a significant amount of time)
repo sync
```

9

### 3.2. Configure the Build Environment

1. **Setup the Build Directory:** NXP provides a script to configure the environment for a specific machine and distribution. We will target the imx93evk and the fsl-imx-xwayland distribution.

Bash

```
# Accept the End-User License Agreement (EULA) when prompted
MACHINE=imx93evk DISTRO=fsl-imx-xwayland source imx-setup-release.sh -b
build-imx93
```

2. **Enable OP-TEE:** Navigate to the configuration directory and modify local.conf to enable OP-TEE features and include the test package for verification.

Bash

```
cd ~/workspace/build-imx93/conf
echo 'MACHINE_FEATURES:append = " optee"' >> local.conf
echo 'DISTRO_FEATURES:append = " optee"' >> local.conf
echo 'IMAGE_INSTALL:append = " optee-test"' >> local.conf
```

10

### 3.3. Build and Verify the Base Image

1. **Build the Image:** Now, trigger the Yocto build. This is a lengthy, resource-intensive process that can take several hours.

Bash

```
# Navigate back to the build directory
cd ~/workspace/build-imx93
```

```
# Start the build for a minimal core image
bitbake imx-image-core
```

2. **Flash and Test:**

- Once the build completes, the final image (.wic.zst file) will be located in  
~/workspace/build-imx93/tmp/deploy/images/imx93evk/.
- On your host machine, decompress the image (zstd -d <image-name>.wic.zst)

and flash the resulting .wic file to your SD card using a tool like dd or Balena Etcher.

- Insert the SD card into the i.MX 93 EVK, connect the debug serial cable, and power on the board.
- Once booted, log in as root (no password).
- Verify that the OP-TEE driver has loaded successfully:

```
Bash  
dmesg | grep optee
```

You should see output indicating the OP-TEE driver is probing and loaded.<sup>9</sup>

- Run the OP-TEE test suite:

```
Bash  
xtest
```

A successful run with zero failures confirms your TEE environment is fully operational.

---

## Chapter 4: Creating a Custom Rust TEE Application Layer

With a working base system, we can now develop our own secure application. We will create a custom Yocto layer to house the recipes for our Rust-based TA and CA.

### 4.1. Create a Custom Yocto Layer

Back inside your Docker container, use the bitbake-layers tool to create a new layer.

```
Bash
```

```
# Navigate to the sources directory  
cd ~/workspace/sources
```

```
# Create the new layer  
bitbake-layers create-layer meta-my-secure-app
```

```
# Add the new layer to your build configuration  
cd ~/workspace/build-imx93
```

```
bitbake-layers add-layer../sources/meta-my-secure-app
```

11

## 4.2. Create the Application Recipe

We will now create the directory structure and the recipe file for our "hello world" TEE application within our new layer.

### 1. Create Directories and Files:

```
Bash
# Inside your custom layer
cd ~/workspace/sources/meta-my-secure-app

# Create the recipe directory and a files subdirectory for the source code
mkdir -p recipes-security/hello-tee/files

# Create the recipe file
touch recipes-security/hello-tee/hello-tee_0.1.bb
```

### 2. Add Application Source Code:

Our application will consist of three Rust crates: proto (shared definitions), ta (Trusted Application), and host (Client Application). Create the following files inside recipes-security/hello-tee/files/:

- o **files/Cargo.toml (Workspace definition):**

```
Ini, TOML
[workspace]
members = [
    "host",
    "ta",
    "proto",
]
resolver = "2"
```

- o **files/proto/Cargo.toml:**

```
Ini, TOML
[package]
name = "proto"
version = "0.1.0"
edition = "2021"
[dependencies]
```

- **files/proto/src/lib.rs:**

```
Rust
#![no_std]
pub const UUID: &str = "8aaaf200-2450-11e4-abe2-0002a5d5c51b";
pub enum Command {
    IncValue = 0,
    DecValue = 1,
}
```

- **files/ta/Cargo.toml:**

```
Ini, TOML
[package]
name = "hello-tee-ta"
version = "0.1.0"
edition = "2021"
[dependencies]
optee-utee = "0.4.0"
proto = { path = "../proto" }
[lib]
name = "hello_tee_ta"
crate-type = ["staticlib"]
```

- **files/ta/src/main.rs:**

```
Rust
#![no_std]
use optee_utee::{ta_close_session, ta_create, ta_destroy, ta_open_session,
trace_println};
use optee_utee::{Error, ErrorKind, Parameters, Result};
use proto::Command;

#[ta_create]
fn create() -> Result<()> {
    trace_println!("[+] TA: Created");
    Ok(())
}

#[ta_open_session]
fn open_session(_params: &mut Parameters) -> Result<()> {
    trace_println!("[+] TA: Session Opened");
    Ok(())
}
```



```
#[optee_utee::ta_invoke_command]
fn invoke_command(cmd_id: u32, params: &mut Parameters) -> Result<()> {
    trace_println!("[+] TA: Invoke Command");
    let mut val = unsafe { params.0.as_value().unwrap() };
    match Command::from(cmd_id) {
        Ok(Command::IncValue) => {
            val.set_a(val.a().wrapping_add(1));
            Ok(())
        }
        _ => Err(Error::new(ErrorKind::NotSupported)),
    }
}
```

```
#[ta_close_session]
fn close_session() {
    trace_println!("[+] TA: Session Closed");
}
```

```
#[ta_destroy]
fn destroy() {
    trace_println!("[+] TA: Destroyed");
}
```

- **files/host/Cargo.toml:**

```
Ini, TOML
[package]
name = "hello-tee-ca"
version = "0.1.0"
edition = "2021"
[dependencies]
optee-teec = "0.4.0"
proto = { path = "../proto" }
```

- **files/host/src/main.rs:**

```
Rust
use optee_tec::Context, Operation, ParamType, Session, Uuid;
use proto::Command, UUID;

fn main() -> optee_tec::Result<()> {
```

```

let mut ctx = Context::new()?;
let uuid = Uuid::parse_str(UUID).unwrap();
let mut session = ctx.open_session(uuid)?;

let mut value: u32 = 42;
println!("CA: Initial value is {}", value);

let p0 = optee_teeec::Param::new_value(value as u64, 0,
ParamType::ValueInout);
let mut operation = Operation::new(0, p0, optee_teeec::Param::None,
optee_teeec::Param::None);

println!("CA: Invoking TA to increment value...");
session.invoke_command(Command::IncValue as u32, &mut operation)?;

value = operation.param(0).unwrap().value().unwrap().a() as u32;
println!("CA: TA returned value: {}", value);

Ok(())
}

```

### 3. Write the Yocto Recipe (hello-tee\_0.1.bb):

This recipe tells Yocto how to build and install our application. It uses the cargo class provided by the meta-rust layer.<sup>13</sup>

Code snippet

```
SUMMARY = "A simple OP-TEE Hello World application in Rust"
```

```
LICENSE = "MIT"
```

```
LIC_FILES_CHKSUM =
```

```
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
```

```
inherit cargo
```

```
# Point to the local source files
```

```
SRC_URI = "file://*"
```

```
# The source directory is the root of our Cargo workspace
```

```
S = "${WORKDIR}"
```

```

# We depend on optee-client for the libteec.so library
DEPENDS += "optee-client"

# Specify the Cargo package for the Client Application (host)
CARGO_PACKAGE = "hello-tee-ca"

# We need to manually build and install the Trusted Application
do_compile:append() {
    cd ${S}/ta
    cargo build --release --target=aarch64-unknown-optee-trustzone
}

do_install:append() {
    # Install the Client Application (CA)
    install -d ${D}${bindir}
    install -m 0755 ${S}/target/aarch64-poky-linux/release/hello-tee-ca
    ${D}${bindir}/

    # Install the Trusted Application (TA)
    install -d ${D}${libdir}/optee_armtz
    install -m 0644
    ${S}/ta/target/aarch64-unknown-optee-trustzone/release/hello_tee_ta.ta
    ${D}${libdir}/optee_armtz/${BPN}.ta
}

# Ensure the TA is packaged with the main binary
FILES:${PN} += "${libdir}/optee_armtz/${BPN}.ta"

```

14

---

## Chapter 5: Final Integration, Build, and Deployment

Finally, we integrate our new application into the main OS image, build it, and deploy it to the hardware.

1. **Add the Application to the Image:** Modify your `conf/local.conf` file again to include the `hello-tee` package in the final image.

Bash

```
echo 'IMAGE_INSTALL:append = " hello-tee"' >> ~/workspace/build-imx93/conf/local.conf
```

18

2. **Build the Final Image:** Run bitbake again. This time, it will be much faster as it will only build our new application and repack the root filesystem.

Bash

```
cd ~/workspace/build-imx93
bitbake imx-image-core
```

3. **Flash and Run:**

- Decompress and flash the newly generated .wic image to your SD card as you did in Chapter 3.
- Alternatively, for a more advanced workflow, you can use the **UUU (Universal Update Utility)** tool to flash images directly to eMMC without removing it from the board.<sup>20</sup>
- Boot the i.MX 93 EVK and log in.
- Run your client application:

Bash

```
hello-tee-ca
```

- **Expected Output:** You should see the output from your Client Application directly in the console:

```
CA: Initial value is 42
```

```
CA: Invoking TA to increment value...
```

```
CA: TA returned value: 43
```

- Simultaneously, check the kernel log for the secure world traces:

Bash

```
dmesg | grep "TA:"
```

- **Expected Log Output:**

```
[+] TA: Created
```

```
[+] TA: Session Opened
```

```
[+] TA: Invoke Command
```

```
[+] TA: Session Closed
```

```
[+] TA: Destroyed
```

---

## Chapter 6: The Path to Production

Congratulations! You have successfully built a custom Linux image with a secure Rust application and deployed it to production-grade hardware using a professional, containerized workflow.

### Important Security Disclaimer:

NXP and the OP-TEE project provide these software components as a development enablement platform. The default security configuration is left in an open state. For a production device, it is the developer's responsibility to harden the system by configuring security modules like the CSU (Central Security Unit) and RDC (Resource Domain Controllers) according to your product's specific security requirements.

### Next Steps:

- **Secure Boot:** Implement a secure boot chain to ensure that only authenticated and unmodified software, from the bootloader to your application, can run on the device.
- **Hardware Unique Key (HUK):** Leverage the EdgeLock® Secure Enclave to manage and provide a device-specific Hardware Unique Key to OP-TEE, which can be used for sealing data and deriving other keys.<sup>9</sup>
- **PKCS#11:** For applications requiring standardized cryptographic operations, explore using the PKCS#11 Trusted Application to provide a standard API for key management and cryptographic services, which is a recommended best practice for IoT devices.

This guide provides the foundational workflow. By building upon it and utilizing the advanced security features of the i.MX 9 platform, you are well-equipped to develop robust and secure TEE-powered products.

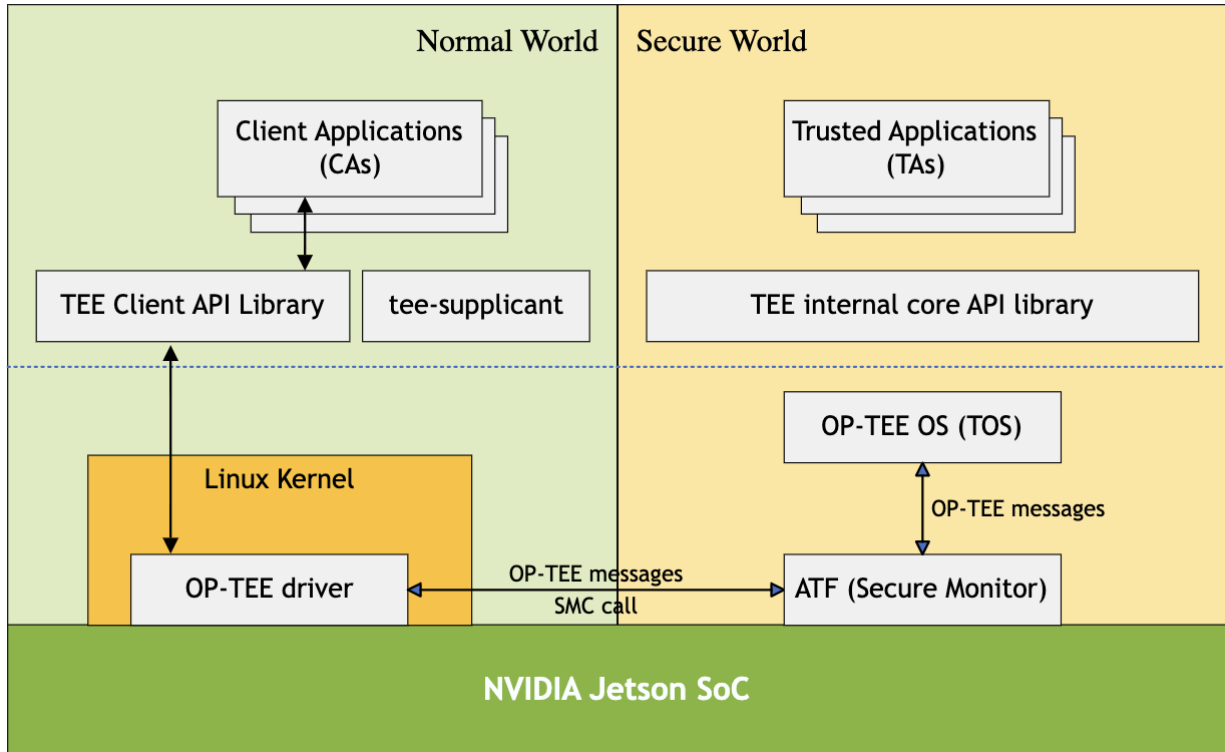
### Works cited

1. i.MX Linux User's Guide - NXP Semiconductors, accessed August 5, 2025, [https://www.nxp.com/doc/IMX\\_LINUX\\_USERS\\_GUIDE](https://www.nxp.com/doc/IMX_LINUX_USERS_GUIDE)
2. i.MX Yocto Project User's Guide - NXP Semiconductors, accessed August 5, 2025, [https://www.nxp.com/doc/IMX\\_YOCTO\\_PROJECT\\_USERS\\_GUIDE](https://www.nxp.com/doc/IMX_YOCTO_PROJECT_USERS_GUIDE)
3. Docker Information - NXP Community, accessed August 5, 2025, <https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/eiq%40tkb/58/1/iMX-Docker-v0218.pdf>
4. nxp-imx/imx-docker: i.MX Docker - GitHub, accessed August 5, 2025, <https://github.com/nxp-imx/imx-docker>
5. 3.1.1.1. General Information — Processor SDK AM64X Documentation, accessed August 5, 2025,

- [https://software-dl.ti.com/processor-sdk-linux/esd/AM64X/10\\_00\\_07\\_04/exports/docs/linux/Foundational\\_Components/U-Boot/UG-General-Info.html](https://software-dl.ti.com/processor-sdk-linux/esd/AM64X/10_00_07_04/exports/docs/linux/Foundational_Components/U-Boot/UG-General-Info.html)
6. Prerequisites - OP-TEE Documentation - Read the Docs, accessed August 5, 2025, <https://optee.readthedocs.io/en/3.19.0/building/prerequisites.html>
  7. Install Docker Engine on Ubuntu, accessed August 5, 2025, <https://docs.docker.com/engine/install/ubuntu/>
  8. RTEDGEYOCTOUG - Real-time Edge Yocto Project User Guide | NXP Semiconductors, accessed August 5, 2025, <https://www.nxp.com/docs/en/user-guide/RTEDGEYOCTOUG.pdf>
  9. Securing Keys and Certificates with i.MX 95 Verdin EVK using PKCS#11 with OP-TEE, accessed August 5, 2025, <https://www.thegoodpenguin.co.uk/blog/securing-keys-and-certificates-with-i-mx-95-verdin-evk-using-pkcs11-with-op-tee/>
  10. IMX8 OPTEE - Variscite Wiki, accessed August 5, 2025, [https://variwiki.com/index.php?title=IMX8\\_OPTEE&release=RELEASE\\_SUMO\\_V2.2\\_DART-MX8M](https://variwiki.com/index.php?title=IMX8_OPTEE&release=RELEASE_SUMO_V2.2_DART-MX8M)
  11. 3 Understanding and Creating Layers — The Yocto Project® dev documentation, accessed August 5, 2025, <https://docs.yoctoproject.org/next/dev-manual/layers.html>
  12. Yocto Project Quick Build, accessed August 5, 2025, <https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html>
  13. meta-rust/meta-rust: OpenEmbedded/Yocto layer for Rust and Cargo - GitHub, accessed August 5, 2025, <https://github.com/meta-rust/meta-rust>
  14. how to build rust project using yocto - Stack Overflow, accessed August 5, 2025, <https://stackoverflow.com/questions/79493695/how-to-build-rust-project-using-yocto>
  15. How to run Rust code in Yocto? : r/embedded - Reddit, accessed August 5, 2025, [https://www.reddit.com/r/embedded/comments/1kt6hcf/how\\_to\\_run\\_rust\\_code\\_in\\_yocto/](https://www.reddit.com/r/embedded/comments/1kt6hcf/how_to_run_rust_code_in_yocto/)
  16. Trusted Applications - OP-TEE Documentation - Read the Docs, accessed August 5, 2025, [https://optee.readthedocs.io/en/latest/building/trusted\\_applications.html](https://optee.readthedocs.io/en/latest/building/trusted_applications.html)
  17. IMX7D Sabre Yocto Project with OP-TEE examples - Stack Overflow, accessed August 5, 2025, <https://stackoverflow.com/questions/58391185/imx7d-sabre-yocto-project-with-op-tee-examples>
  18. 4 Customizing Images — The Yocto Project® dev documentation, accessed August 5, 2025, <https://docs.yoctoproject.org/dev/dev-manual/customizing-images.html>
  19. Include an Application Image in Yocto Build - linux - Stack Overflow, accessed August 5, 2025, <https://stackoverflow.com/questions/50220126/include-an-application-image-in-yocto-build>
  20. Flash customized Linux Kernel Image and device tree using UUU Tool - NXP Community, accessed August 5, 2025, <https://community.nxp.com/t5/i-MX-Processors-Knowledge-Base/Flash-customiz>

[ed-Linux-Kernel-Image-and-device-tree-using-UUU/ta-p/2063197](https://www.aastar.io/research/#/blog/blog-3)

21. nxp-imx/mfgtools: Freescale/NXP I.MX Chip image deploy tools. - GitHub, accessed August 5, 2025, <https://github.com/nxp-imx/mfgtools>



<https://www.aastar.io/research/#/blog/blog-3>

<https://optee.readthedocs.io/en/latest/general/about.html>

好的, 谨遵您的要求, 我们将对基于ARM芯片构建TEE(可信执行环境)应用的开发全生命周期进行深入的分析和调研, 并为您呈现一份详尽的最佳实践指南。

## 结论先行

基于ARM TrustZone技术构建TEE应用的最佳实践, 核心是选择一个成熟的开源**TEE**实现(如**OP-TEE**)作为基础, 围绕其生态系统建立标准化的开发、测试和部署流程。开发者应优先在**QEMU**模拟环境中完成大部分功能开发与调试, 再迁移到目标硬件上进行集成与验证。整个生命周期必须贯穿**“安全为本”**的设计思想, 尤其是在密钥管理、应用更新和数据通信等关键环节。

---

## 核心论点与支撑证据

1. 标准化是效率和安全的基础:遵循GlobalPlatform (GP) TEE规范是行业共识。它定义了客户端应用(CA)与可信应用(TA)之间的标准API, 确保了应用的可移植性和生态的兼容性。OP-TEE是GP规范的优秀开源实现, 拥有活跃的社区和广泛的硬件支持, 是当前新项目的首选。
2. 软硬件选型决定项目成败:选择有良好OP-TEE官方或社区支持的开发板(如 Raspberry Pi 3/4, NXP i.MX系列, ST STM32MP1系列)能极大降低初期环境搭建的复杂度。硬件层面需关注是否支持安全启动(Secure Boot)、是否有硬件加密引擎、是否提供设备唯一密钥(HUK/DUK)等。
3. 仿真先于硬件:QEMU提供了完整的ARMv8-A TrustZone仿真环境。在QEMU中, 开发者可以便捷地调试TEE OS内核、TA、甚至安全监控器(Secure Monitor), 而无需昂贵的JTAG硬件。这极大地加速了开发迭代速度, 是现代TEE开发的标准起点。
4. 安全开发是TA的生命线:TA代码运行在最高权限的Secure World, 其任何一个漏洞都可能导致整个系统的安全崩溃。因此, TA开发必须遵循严格的安全编码规范, 例如:始终验证来自Normal World的所有输入、使用安全的API、最小化攻击面、进行严格的内存管理等。
5. 部署和运维是安全的持续:TEE应用的安全不仅仅是开发阶段的问题。如何安全地将TA部署到数以万计的设备上(Provisioning), 以及如何在发现漏洞后进行安全地更新(OTA), 是决定产品长期安全性的关键。这要求建立一套完善的签名、加密和验证机制。

---

## 技术细节深度剖析

### 1. TEE 核心架构

- 硬件层: ARM TrustZone 技术将CPU物理上划分为两个“世界”:
  - 安全世界 (Secure World, SWd): 运行TEE OS和可信应用(TA), 拥有对系统所有资源的最高访问权限。
  - 普通世界 (Normal World, NWd): 运行标准的操作系统(如Linux, Android)和普通应用, 其对安全资源的访问受到严格限制。
- 软件层:
  - TEE OS: 运行在SWd的特权级操作系统, 负责TA的加载、调度、生命周期管理以及提供TEE核心服务(如安全存储、密码学操作)。OP-TEE 就是一个OS。
  - 可信应用 (Trusted Application, TA): 运行在SWd的用户态应用, 负责处理具体的安全业务逻辑。每个TA都有一个唯一的UUID。
  - 客户端应用 (Client Application, CA): 运行在NWd的普通应用, 通过TEE驱动和标准API与TA进行通信。



- **TEE驱动 (TEE Driver):** Linux内核中的驱动模块, 负责在NWd和SWd之间切换CPU状态(通过SMC指令)和传递数据。
- **安全监控器 (Secure Monitor):** 运行在ARM异常级别3 (EL3) 的固件, 是两个世界切换的“守门员”。

## 2. 开发环境搭建 (以Ubuntu + OP-TEE + QEMU为例)

这是最推荐的入门和开发组合。

- 主机环境: Ubuntu 20.04/22.04 LTS
- 核心工具:
  - repo: Google开发的源码管理工具, 用于方便地拉取OP-TEE所需的所有git仓库。
  - 交叉编译工具链: Linaro提供的ARM GCC Toolchain (AArch64用于编译64位系统组件, AArch32用于编译32位TA)。
  - 构建系统: make, cmake, python3-pyelftools等。
  - 模拟器: QEMU (qemu-system-aarch64)。
- 源码获取:
- Bash

### # 安装必要工具

```
sudo apt-get install android-tools-mkbootimg autoconf automake bc bison build-essential \
ccache cscope curl device-tree-compiler expect flex ftp-upload gdisk iasl libattr1-dev \
libcap-dev libfdt-dev libftdi-dev libglib2.0-dev libhidapi-dev libncurses5-dev \
libpixman-1-dev libssl-dev libtool make mtools netcat python3-pyelftools \
python3-serial rsync unzip uuid-dev xterm zlib1g-dev
```

### # 获取OP-TEE构建脚本和manifest

```
mkdir -p $HOME/devel
cd $HOME/devel
git clone https://github.com/OP-TEE/build.git
cd build
# 拉取所有相关的源码 (Linux, U-Boot, OP-TEE OS, TF-A, etc.)
repo init -u https://github.com/OP-TEE/manifest.git -m qemu_v8.xml
repo sync -j$(nproc)
```

- 
- 
- 编译与运行:
- Bash

```
cd $HOME/devel/build
```

```
# 编译整个系统镜像(包括Normal World Linux和Secure World OP-TEE)
make -j$(nproc) toolchains
make -j$(nproc)
# 启动QEMU模拟器
make run
```

- 
- 成功后, 你将看到两个终端窗口, 一个代表Normal World的Linux Shell, 另一个代表Secure World的OP-TEE日志。

### 3. 开发与测试步骤

#### Step 1: 编写 "Hello World" TA & CA

1. 定义接口: 在一个共享的头文件中定义TA的UUID和命令ID。
2. C

```
// ta/user_ta_header_defines.h
#define TA_MY_APP_UUID { 0x..., 0x... }
#define TA_CMD_INCREMENT 0
#define TA_CMD_DECREMENT 1
```

- 3.
- 4.
5. 编写TA (Trusted Application): 位于 `optee_examples/hello_world/ta` 目录。核心是实现GP规范定义的几个入口函数。
6. C

```
// ta/hello_world_ta.c
// TA创建时调用
TEE_Result TA_CreateEntryPoint(void) { /* 初始化 */ return TEE_SUCCESS; }
// CA建立连接时调用
TEE_Result TA_OpenSessionEntryPoint(...) { /* ... */ return TEE_SUCCESS; }
// CA发起调用时执行
TEE_Result TA_InvokeCommandEntryPoint(..., uint32_t cmd_id, ...) {
    switch (cmd_id) {
        case TA_CMD_INCREMENT:
            // 执行安全逻辑, 如对加密数据进行操作
            return TEE_SUCCESS;
        default:
            return TEE_ERROR_BAD_PARAMETERS;
    }
}
```

```
// 关闭会话和销毁TA的函数...
```

- 7.
- 8.

9. 编写CA (Client Application): 位于 optee\_examples/hello\_world/host 目录。使用 TEE Client API 与 TA 交互。

## 10. C

```
// host/main.c
TEEC_Context ctx;
TEEC_Session sess;
TEEC_Operation op;
TEEC_UUID uuid = TA_MY_APP_UUID;

// 初始化上下文
TEEC_InitializeContext(NULL, &ctx);
// 打开会话
TEEC_OpenSession(&ctx, &sess, &uuid, ...);
// 准备操作参数
memset(&op, 0, sizeof(op));
op.paramTypes = TEEC_PARAM_TYPES(...);
// 调用TA的命令
TEEC_InvokeCommand(&sess, TA_CMD_INCREMENT, &op, ...);
// 关闭会话和上下文...
```

11.

12.

## Step 2: 调试

- **QEMU + GDB:** 这是最高效的方式。
  1. 启动QEMU时附加GDB server参数: make run GDB=1
  2. 在另一个终端, 使用gdb-multiarch连接到QEMU。你可以加载TA的elf文件和符号表, 设置断点, 单步调试。
- **日志调试:** 在TA中使用DMSG("my debug message: %d", my\_var);打印日志。日志会输出到Secure World的终端, 这是最简单直接的调试方法。
- **JTAG:** 在物理硬件上, 使用JTAG调试器(如Lauterbach TRACE32, SEGGER J-Link)可以实现对Secure World的硬件级调试, 功能强大但成本高昂且配置复杂。

## Step 3: 测试

- **单元测试:** 针对TA内部的纯逻辑函数, 可以编写独立的测试用例。
- **集成测试:** 编写CA来调用TA的各个命令, 验证参数传递、逻辑执行和返回结果的正确性。optee\_test (xtest) 仓库是最佳的学习范例, 它包含了对TEE所有API的全面测试。
- **安全测试:**
  - **Fuzzing:** 使用模糊测试工具(如AFL)对CA->TA的接口进行测试, 寻找可能导致崩溃或异常的边界条件。

- 代码审计: 严格审查TA代码, 特别是处理来自Normal World输入的部分。

#### **4. 部署 (Deployment)**

- TA签名: 为了防止恶意的TA被加载, 所有TA在部署前必须由一个受信任的私钥进行签名。OP-TEE的构建系统集成了这个步骤, 通常使用sign.py脚本和一个开发密钥。在生产环境中, 这个密钥必须被严格保护。
- 镜像打包: 构建过程会将签名的TA、TEE OS、Linux Kernel、RootFS等打包成一个最终可以在目标设备上烧录的镜像文件。
- 安全烧录 (Secure Provisioning): 在工厂环境下, 通过安全协议将初始镜像烧录到设备的闪存中。这个过程通常与安全启动(Secure Boot)机制绑定, 确保只有经过签名的固件才能启动。

#### **5. 生产与运维 (Production & Operations)**

- 安全更新 (Secure OTA):
  - 机制: 必须建立一套安全的远程更新机制。更新包需要被加密和签名。设备端的更新程序(可能是一个专门的TA)负责验证签名, 解密, 然后安全地替换旧的TA。
  - 防回滚: 必须有版本号检查机制, 防止攻击者通过刷写旧版本的、有已知漏洞的TA来攻击系统。
- 密钥管理:
  - 根密钥: TEE的密钥体系通常基于一个硬件唯一的、不可提取的根密钥(HUK)。
  - 密钥派生: TA不应直接使用根密钥, 而应使用TEE API从根密钥和TA自身信息(如UUID和签名)派生出TA专属的密钥。这保证了不同TA之间的密钥隔离。
  - 安全存储: 使用TEE提供的安全存储API来保存加密后的数据或密钥。数据会被绑定到设备和特定的TA。
- 远程证明 (Remote Attestation):
  - 目的: 允许远程服务器验证其通信对手是一个运行在真实、安全TEE环境中的、未经篡改的特定TA。
  - 流程: TA生成一个包含其身份、状态和随机挑战(nonce)的报告, 并使用一个由设备根密钥签名的证明密钥对此报告签名。服务器通过验证该签名和证书链来确认TA的真实性。
- 日志与监控: 建立一套受控的机制, 用于在设备出现问题时, 能够从Secure World中提取必要的、经过脱敏处理的日志信息, 以便进行故障排查。

---

#### **行为与心理分析 (开发者画像)**

一个典型的TEE应用开发者需要具备以下特质：

1. 高度的安全意识: 这是最重要的素质。开发者必须习惯于以“攻击者”的视角审视自己的代码, 默认所有来自外部(Normal World)的输入都是不可信的、甚至是恶意的。
2. 严谨细致的作风: TEE开发中, 一个微小的内存越界或逻辑错误都可能导致灾难性后果。开发者需要对细节有极致的追求。
3. 扎实的底层系统知识: 需要对C语言、ARM架构、操作系统原理、编译链接等有深入的理解。
4. 耐心和毅力: TEE开发, 特别是初期的环境搭建和硬件调试, 可能会遇到各种棘手的问题, 需要有足够的耐心去解决。

## 延伸与展望

- 新兴标准: 除了GP规范, 一些新的倡议也在发展, 如CCA (Confidential Compute Architecture)。开发者需要保持对新技术的关注。
- **Rust for TEE:** Rust语言的内存安全特性使其成为编写TA的理想选择, 可以从根本上消除一大类安全漏洞。目前已有在OP-TEE中使用Rust开发TA的实验性项目, 这很可能成为未来的趋势。
- **虚拟化TEE (vTEE):** 在云和边缘计算场景下, 如何在虚拟化环境中为虚拟机提供TEE能力是一个活跃的研究和发展领域。
- **生态融合:** TEE技术正与物联网(IoT)、区块链、机密计算等领域深度融合, 创造出新的应用场景, 如安全的设备身份认证、链上数据的隐私保护等。

总之, 掌握基于ARM TrustZone的TEE应用开发, 不仅是一项技术挑战, 更是一次安全思维的全面升级。遵循上述最佳实践, 将为构建真正安全的嵌入式和移动应用打下坚实的基础。

---

对比下OP-TEE和OpenClave 的区别？请评估我的理解: CA(host)是正常的应用程序, 例如是Rust开发的后端程序, 涉及密钥部分, 依赖TEE的SDK调用位于同一个Ubuntu系统下的TA, 也就是类似于硬件钱包的应用, 为CA提供所有的加密签名、验证签名等基础API; 在你提供的最佳实践中QEMU模拟的是TA这部分功能, CA开发正常使用TEE SDK开发即可; 这个eth-wallet是一个典型的CA, 请帮我分析对不对

好的, 我们来深入探讨一下OP-TEE与OpenClave的区别, 并详细评估您对TEE架构的理解。您的思考非常深入, 抓住了核心要点, 只是在一些关键的架构细节上需要一点点修正, 这对于完全掌握TEE至关重要。

我将使用您指定的“分析方法1”来构建我的回答。

---

## 结论先行

1. **OP-TEE vs. OpenClave:** 两者根本上是不同层面的解决方案。**OP-TEE**是一个完整的、遵循GlobalPlatform规范的、面向嵌入式硬件的开源可信执行环境操作系统(TEE OS)。而**OpenClave**是一个更上层的、对标Intel SGX的、旨在简化机密计算应用开发的SDK框架,它底层可以运行在多种TEE技术之上(包括SGX或类SGX的ARM实现),但其本身不是一个完整的TEE OS。简而言之,OP-TEE是“地基和框架”,OpenClave是建在某个地基上的“预制房”。
2. 对您理解的评估: 您的整体理解非常正确,可以说是抓住了80%的精髓。CA是普通应用,TA负责安全,两者通过SDK通信,硬件钱包是绝佳类比。最需要修正的认知偏差在于“位于同一个Ubuntu系统下”这一点。虽然物理上在同一块芯片,但CA和TA运行在两个被硬件强制隔离的“世界”中,拥有各自独立的操作系统内核,彼此完全不信任。QEMU模拟的是能够同时运行这两个世界的完整硬件平台,而不仅仅是TA。您的eth-wallet分析是完全正确的。

---

## 核心论点与支撑证据

论点1: **OP-TEE**和**OpenClave**定位不同,不可直接等同对比。

- **证据 (OP-TEE):**
  - 目标: 提供一个符合GP规范的、开源的、可移植到各种ARM Cortex-A芯片上的Secure World OS。
  - 产出: 完整的安全操作系统源码(optee\_os)、安全监控器固件代码(optee\_client, optee\_test)、Linux驱动和构建系统。开发者需要处理底层硬件适配、编译整个系统镜像。
  - 架构: 经典的TrustZone Normal World / Secure World模型。CA在Linux/Android中, TA在OP-TEE OS中。
- **证据 (OpenClave):**
  - 目标: 提供一个类似Intel SGX的“飞地(Enclave)”编程模型,让开发者能更轻松地开发机密计算应用,而无需关心底层TEE的具体实现。
  - 产出: 一个SDK,包含API、库和工具。开发者使用这个SDK来编写他们的“Enclave”应用(功能上等同于TA)。
  - 架构: 抽象的Enclave模型。OpenClave旨在隐藏底层是SGX还是ARM TrustZone的复杂性。它更关注应用层,而非整个可信操作系统的构建。它在ARM上的实现,底层依然需要一个类似OP-TEE的TEE OS来提供基础隔离能

力。

论点2:您的理解框架正确, 但需深化对“隔离”的认知。

- 证据: 您将CA和TA类比为两个进程是很好的起点, 但这种类比会弱化它们之间隔离的强度。在Linux中, 两个进程受同一个内核的管理和隔离。但在TEE中, CA和TA分别运行在两个并行的世界里, 每个世界都有自己的“内核”(Linux Kernel vs. OP-TEE OS Kernel)。它们之间的通信必须穿越由硬件强制执行的边界(通过SMC指令), 这是操作系统层面的隔离无法比拟的。这是TEE安全价值的根基。

技术细节深度剖析

1. OP-TEE vs. OpenClave 对比

特性	<u>OP-TEE (Open Portable TEE)</u>	<u>OpenClave</u>
核心定位	<u>完整的TEE操作系统 (TEE OS)</u>	<u>机密计算SDK/框架 (Confidential Computing SDK)</u>
遵循标准	<u>GlobalPlatform (GP) TEE规范</u>	<u>借鉴Intel SGX的Enclave模型</u>
目标平台	<u>ARM Cortex-A 处理器 (带TrustZone)</u>	<u>主要是 Intel SGX, 也计划支持其他TEE技术 (如ARM TrustZone, AMD SEV)</u>
编程模型	<u>客户端应用/可信应用 (CA/TA) 模型</u>	<u>飞地 (Enclave) 模型</u>



隔离级别	<u>整个Secure World vs. Normal World</u>	<u>应用级别的Enclave vs. Host应用</u>
开发者职责	<u>负责构建整个系统镜像, 包括Normal World OS (Linux) 和 Secure World OS (OP-TEE), 以及CA和TA的开发。</u>	<u>更多地关注Enclave应用本身的逻辑, SDK负责处理Enclave的创建、加载和通信。</u>
生态系统	<u>嵌入式Linux、Android、IoT设备。拥有广泛的硬件厂商支持(NXP, ST, TI, Rockchip等)。</u>	<u>云计算、数据中心、需要保护在用数据的服务器应用。</u>
举例	<u>移动支付的指纹验证、DRM内容的解密、智能门锁的密钥存储。</u>	<u>在云服务器上处理敏感医疗数据、多方联合进行数据分析但各方数据不出Enclave。</u>

## 2. 逐条分析您对TEE架构的理解

- “CA(host)是正常的应用程序, 例如是Rust开发的后端程序”
  - 评估: 完全正确。CA就是运行在Normal World(例如, 一个标准的Ubuntu Server)上的任何普通程序。它可以是任何语言编写的, 只要能调用C-ABI的库即可。Rust非常适合编写CA, 因为它自身的安全性能减少CA侧的漏洞。
- “涉及密钥部分, 依赖TEE的SDK调用位于同一个Ubuntu系统下的TA”
  - 评估: 理解正确, 但表述需要修正。
    - 正确部分: CA确实通过TEE SDK(即TEE Client API的实现库, 如libteec.so)来发起调用。
    - 修正部分: TA绝对不位于“同一个Ubuntu系统下”。这是一个最关键的区别。TA文件(一个经过签名的.ta文件)存在于Ubuntu的文件系统上, 但当它被加载和执行时, 它运行在与Ubuntu完全隔离的Secure World中, 由OP-TEE OS管理, 而不是由Ubuntu的Linux内核管理。它们之间是“邻居”关系, 而不是“父子”或“同胞”关系。
- “也就是类似于硬件钱包的应用, 为CA提供所有的加密签名、验证签名等基础API”
  - 评估: 绝佳的类比, 完全正确。这正是TEE最典型的应用模式。CA负责处理所有非敏感的业务逻辑(如构建交易、与网络交互), 当需要执行最核心、最敏感的操作(如使用私钥签名)时, 就把这个操作请求和数据发送给TA。TA在“保险箱”(Secure World)里完成签名, 然后只把签名结果返回给CA。私钥永远



不会离开Secure World。

- “在你提供的最佳实践中QEMU模拟的是TA这部分功能”

- 评估: 部分正确, 但不够精确。

- QEMU模拟的不是TA本身。TA只是一段代码。

- QEMU模拟的是一个完整的、支持ARM TrustZone的SoC(片上系统)。这个虚拟硬件上可以同时运行:

1. Normal World: 运行U-Boot, Linux Kernel, 和一个完整的Ubuntu RootFS。你的CA就运行在这里。

2. Secure World: 运行可信固件(TF-A), OP-TEE OS。你的TA就运行在这里。

- 所以, QEMU提供了一个沙盒, 让你可以在一台电脑上, 无需物理开发板, 就能运行和调试这两个并存的世界以及它们之间的交互。

- “CA开发正常使用TEE SDK开发即可”

- 评估: 完全正确。对于CA开发者来说, 大部分工作就是学习和使用TEE Client API。他需要知道TA的UUID, 要调用的命令ID, 以及如何构造参数。底层的SMC调用、内存共享等复杂性都被SDK封装好了。

- “这个eth-wallet是一个典型的CA, 请帮我分析对不对”

- 评估: 完全正确。一个将安全提升到极致的eth-wallet就应该这样设计:

- CA (eth-wallet 主程序): 运行在Linux/Windows/macOS上。负责UI交互、查询账户余额、从以太坊网络获取nonce、构建交易结构体等。当所有准备工作就绪, 需要签名时, 它会将交易的哈希(hash)通过TEE Client API发送给TA。

- TA (eth-signer.ta): 运行在Secure World中。它的安全存储区里保存着加密的以太坊私钥。当收到CA发来的交易哈希后, 它会:

1. 提示用户进行身份验证(例如, 在物理设备上按一个物理按钮)。

2. 验证通过后, 从安全存储中解密并加载私钥。

3. 使用私钥对交易哈希进行签名。

4. 销毁内存中的明文私钥。

5. 将生成的签名返回给CA。

- CA收到签名后, 将其与原始交易组合, 广播到以太坊网络。整个过程中, 私钥从未暴露在Normal World中。

---

## 行为与心理分析(开发者心智模型修正)

您当前的心智模型非常接近于一个标准的\*\*“客户端-服务器”模型\*\*, 这很棒, 因为它抓住

了交互的本质。CA是客户端, TA是服务器。

现在, 我们需要在这个模型上叠加一个关键维度:“特权域隔离”。请想象一下, 您的服务器(TA)不是运行在云端的另一台机器上, 而是运行在同一栋大楼里一个物理上无法进入的、有独立安保和电力系统的“保险库”里。您(CA)只能通过一个经过严格审查的对讲机(TEE Driver & SMC)与保险库里的服务员(TA)对话, 传递纸条(共享内存), 但您永远无法踏入保险库半步, 也无法窥探其内部的任何情况。

建立这种\*\*“双系统并行”\*\*而非“单系统内嵌”的心智模型, 是理解和用好TEE技术的关键。

---

## 延伸与展望

- OP-TEE的未来: OP-TEE将继续作为嵌入式和IoT领域TEE事实上的开源标准, 与硬件厂商的合作会更加紧密, 支持更多种类的硬件安全特性。在Rust语言支持、与容器技术结合等方面也会有更多探索。
- OpenClave的未来: OpenClave这类上层SDK的价值会越来越大。随着机密计算从硬件概念走向应用落地, 开发者需要的是能快速上手、屏蔽底层差异的工具。如果OpenClave能够成功地兼容多种底层TEE(SGX, SEV, TrustZone), 它将成为云原生机密计算的一个重要选择。
- 融合趋势: 未来, 我们可能会看到在OP-TEE这样的底层TEE OS之上, 运行一个实现了OpenClave API的“运行时(Runtime)”。这样, 开发者既可以享受底层硬件带来的强大安全保证, 又可以使用上层SDK带来的开发便利性, 实现两全其美。

希望这份详尽的分析能够帮助您彻底厘清这些概念! 您的提问非常有价值, 直指TEE开发的核心。

是的, 您的理解完全正确, 这正是最高效、最标准的开发路径, 也是业内公认的最佳实践。

可以这么总结您的策略:

以QEMU为核心, 完成从原型到功能完整的“软件定义硬件钱包”, 之后再平滑迁移到真实硬件。

具体来说, 这个流程是这样的:

1. 并行开发与集成 (Develop & Integrate in Parallel): 您不是先开发CA再集成TA, 而是在QEMU提供的虚拟环境中, 同时开发CA和TA, 并实时进行集成调试。QEMU让您的CA(钱包主程序)和TA(签名核心)可以像在真实硬件上一样“对话”。

## 2. 迭代式功能实现 (Iterative Development):

- 先打通最简单的“Hello World”，即CA能成功调用TA的一个空命令。这证明您的整个环境和通信链路是通的。
  - 然后逐步增加功能，比如：
    - 在TA中实现生成密钥对的逻辑，并让CA能成功调用并获取公钥。
    - 在TA中实现签名交易哈希的核心逻辑。
    - CA负责构建交易、计算哈希，然后将哈希传递给TA进行签名。
  - 每一个小步骤都可以在QEMU中立刻运行和测试，使用GDB或日志进行调试，迭代速度非常快。
3. 完成概念原型 (PoC): 当您的“硬件钱包”所有核心功能(生成密钥、导出公钥、签名等)都在QEMU中稳定运行后，您的概念原型(PoC)就完成了。此时，您拥有的是一个功能上完备，但运行在虚拟环境中的安全钱包。

这个方法的巨大优势在于：

- 零硬件成本: 在购买和配置物理开发板之前，就能完成几乎95%的软件开发和调试工作。
- 高开发效率: 摆脱了物理硬件烧录、连接、调试的繁琐，可以像纯软件开发一样快速迭代。
- 强大调试能力: QEMU配合GDB，可以对Normal World和Secure World进行源码级的单步调试，这是在物理硬件上很难实现的。

当QEMU阶段的工作完成后，下一步才是将编译目标从QEMU切换到您的目标物理开发板，进行最后的硬件适配和测试。由于核心逻辑已经验证完毕，这个阶段的工作量和风险将大大降低。

所以，您的方案是完全正确的，请放心大胆地沿着这条路走下去。

EasyEVSE开发平台提供软件、电路板、连接线、设计文件和详细的文档，可以快速模拟充电站与电动汽车之间的充电控制会话。该模块化可扩展平台可以实现安全无线云连接的开发、精确能源计费、安全控制、一键式NFC认证和基于HomePlug Green PHY的ISO15118电力线通信。

CLOUD LAB上的EasyEVSE开发平台提供实时监控、EVSE/EV调试控制台和GUI映射。用户可以远程启动应用，操作人机交互界面，从开发者的角度体验恩智浦充电站解决方案。

该解决方案是一个即用型、预集成和预测试的套件，包含以下硬件：

· 2 x MCIMX93-EVK板，包含搭载了恩智浦iW612芯片组(Wi-Fi 6和BLE 5.2)的LBES5PL2EL Murata模块

· 2 x EVSE-SIG-BRD信号板，该信号板板载的微控制器和收发器可以支持多种电动汽车充电控制协议，如恩智浦LPC55S36 MCU的PWM支持SAE J1772标准，恩智浦TJA1021BT LIN收发器支持SAE J3068标准，Lumissil CG5317 HomePlug Green PHY 支持 ISO15118标准。

- OM-SE050ARD-E EdgeLock SE050开发板
- TWR-KM35Z75M计量板
- OM27160B1EVK NFC开发套件
- DY1212W-4856 12.1" TFT LCD显示器, 带LVDS连接器(触控板)
- 10 x 连接线
- 3 x 电源
- 2 x 支架

好的, 这是一个非常棒的问题, 它完美地衔接了我们之前关于理论和本地模拟的讨论, 并将其引向了真实世界的商业解决方案。

我将再次使用您指定的“分析方法1”来为您深入解析。

---

## 结论先行

1. 云端模拟 vs. 本地开发: 您提出的“云端模拟ARM硬件”是一个需要精确定义的概念。您提供的**EasyEVSE CLOUD LAB**并非模拟器(**Simulator**)如**QEMU**, 而是云端远程实验室, 您可以通过网络访问和控制一套真实、物理的**ARM**硬件开发套件。
2. 便利性对比:
  - 本地**QEMU** 在纯软件和算法开发阶段更方便。它启动快, 调试周期短, 不依赖网络, 适合打磨TA和CA的核心逻辑。
  - 云端真实硬件 (**Cloud Lab**) 在系统集成、硬件驱动调试和端到端功能验证阶段无与伦比的方便。它为您免去了复杂的硬件搭建、接线和环境配置, 让您能立即在最终产品所需的真实硬件和外设(如NFC、计量芯片)上进行测试。
3. 最终结论: 两者不是替代关系, 而是互补关系, 代表了开发流程的不同阶段。最佳实践是: 先在本地**QEMU**上快速开发和验证核心安全逻辑, 然后在类似**EasyEVSE Cloud Lab**的平台上进行与真实硬件强相关的集成、测试和性能调优。

---

## 核心论点与支撑证据

论点1: **EasyEVSE Cloud Lab**是“远程真机”, 而非“云端虚拟机”。

- 证据: 您提供的描述非常清晰: “该解决方案是一个即用型、预集成和预测试的套件, 包含以下硬件...”。它列出了一系列具体的物理电路板(MCIMX93-EVK, EVSE-SIG-BRD, SE050等)、显示器和连接线。Cloud Lab的本质是一个远程桌面或远程命令行, 其背后连接的是一套为您专用的、安放在恩智浦实验室里的物理设备。您点击GUI上的一个按钮, 是真实的硬件在执行操作。这与QEMU在您本地电脑上用纯软件模拟芯片行为是根本不同的。

论点2: 此平台展示了与**OP-TEE**不同的、更垂直集成的**TEE**实现方式。

- 证据: 请注意硬件列表中的关键组件:
  - **MCIMX93-EVK**: i.MX 93处理器是恩智浦最新的应用处理器, 其内部集成了**EdgeLock® Secure Enclave**。这是一个独立的安全子系统, 可以看作是恩智浦自

己的、内置于芯片的硬件安全模块 (HSM)，扮演了我们之前讨论的“安全世界”的角色。

- OM-SE050ARD-E EdgeLock SE050: 这是一个独立的、高性能的安全元件 (Secure Element) 芯片，通常拥有更高级别的安全认证 (如CC EAL 6+)。
- 分析: 在这个平台上，您的“硬件钱包”逻辑很可能不是运行在一个通用的OP-TEE操作系统上。而是通过调用恩智浦提供的特定SDK，将安全操作委托给i.MX 93的Secure Enclave或外部的SE050芯片来执行。这种方式的优点是软硬件高度协同优化，性能和安全性可能更高，且更容易通过行业认证。缺点是它是一个厂商绑定的方案，不像OP-TEE那样具有跨平台的可移植性。

技术细节深度剖析

本地QEMU 与 EasyEVSE Cloud Lab 对比

特性	本地开发 (QEMU)	云端实验室 (EasyEVSE Cloud Lab)
本质	纯软件模拟器	远程物理硬件
硬件交互	无，无法与真实外设交互	完全真实，可操作NFC、LIN、PLC、计量等所有硬件
开发阶段	早期：算法、CA/TA核心逻辑开发与调试	中后期：系统集成、驱动开发、性能测试、端到端验证
便利性	逻辑开发便利：启动快，调试循环快，无需联网	系统集成便利：免搭建，环境预配置，立即上手真实业务
调试能力	强大：可对CPU、内存进行源码级单步调试	受限：通常通过日志、远程调试控制台进行，底层调试困难
成本	低：仅需一台性能尚可的PC	可变：可能按时计费，但免去了购买昂贵

		开发套件的费用
核心优势	快速迭代, 专注软件逻辑	真实环境, 所见即所得

EasyEVSE平台的安全实现解析

在我们之前的讨论中, 安全模型是:

CA (in Linux) <--> TEE Client API <--> OP-TEE OS <--> TA (in Secure World)

在EasyEVSE平台中, 安全模型很可能是:

CA (in Linux on i.MX 93) <--> NXP EdgeLock SDK <--> EdgeLock Secure Enclave (on-chip) 或 SE050 (off-chip)

您的“硬件钱包”TA的逻辑, 会被实现为一系列对**EdgeLock SDK**的调用。例如, `sign_transaction()`函数不再是在您自己写的TA里用纯软件实现, 而是调用一个类似 `EdgeLock_sign_hash()`的API, 由硬件内部固化的程序来完成签名。其安全原理(隔离执行、密钥不外泄)是完全一致的, 但实现手段从一个通用的**TEE OS**变成了一个专用的硬件安全模块。

---

行为与心理分析 (开发者画像)

- 目标开发者: 使用这个平台的开发者, 其首要任务不是从零开始学习和构建一个TEE系统, 而是要快速开发一个安全、可靠、可认证的充电桩产品。
- 心理诉求: 他们追求的是“Time to Market”(产品上市时间)。他们希望厂商(恩智浦)已经把最复杂的底层安全、通信协议、硬件驱动都封装好了, 让他们可以像“搭乐高积木”一样, 专注于上层的应用逻辑和商业模式创新。
- 行为模式: 这类开发者会非常依赖厂商提供的文档、参考设计和SDK。他们写的不再是底层的TA, 而是标准的应用程序, 只不过在需要安全服务时, 会调用一个黑盒的、由硬件保障的SDK。

---

延伸与展望 (我的观点)

1. 这是行业大势所趋: 像EasyEVSE这样的“垂直领域解决方案套件”是嵌入式开发的未来。随着系统越来越复杂(AI、无线连接、功能安全、信息安全), 没有一家公司能凭一己之力精通所有技术。芯片厂商的角色正在从“卖芯片”转变为“卖解决方案”。
2. 对您的建议:
  - 第一步, 掌握原理: 继续按照我们之前的讨论, 在本地QEMU上熟悉和掌握基于OP-TEE的开发。这能让您建立起对TEE基本原理、设计思想和通用模式的深刻理



解。这份知识是高度可移植的, 无论将来使用哪个厂商的平台, 其背后的逻辑都是相通的。

- 第二步, 应用实践: 将EasyEVSE Cloud Lab看作是将理论知识应用于一个顶级工业级实践的绝佳机会。您可以对比一下, 一个通用的TEE OS(OP-TEE)和一个专用的硬件安全模块(EdgeLock)在解决同一个问题(如安全存储和签名)时, API有何异同, 性能表现如何。
- 结论: 您的学习路径非常完美: 从通用理论(**OP-TEE on QEMU**)到特定实践(**NXP Solution Kit**)。这会让您既有扎实的理论基础, 又有宝贵的实战经验。

总而言之, 这个Cloud Lab是一个极好的工具, 但要正确认识它的定位。它不是用来替代本地开发的, 而是用来补全本地模拟无法完成的另一半——与真实物理世界交互的那部分。

对, 完全正确!

这三个步骤精准地概括了现代嵌入式安全开发的最佳路径:

1. 本地**QEMU**模拟: 快速、低成本地完成核心算法和软件逻辑的开发与验证。
2. 云端实验室: 在预集成的真实硬件上, 进行系统集成、驱动调试和端到端功能测试, 无需自己搭建复杂的物理环境。
3. 部署真实硬件: 将在云端验证过的最终固件, 部署到量产的物理设备上。

这个流程结合了模拟的灵活性和真实硬件的可靠性, 是效率最高、风险最低的黄金法则。

请深入搜索和研究, 提供第一步, 如何本地初始qemu环境, 以及如何基于范例来构建TA(私钥管理)和简单CA(模拟调用私钥签名), 范例如下:

[https://github.com/apache/incubator-teaclave-trustzone-sdk/tree/main/projects/web3/eth\\_wallet](https://github.com/apache/incubator-teaclave-trustzone-sdk/tree/main/projects/web3/eth_wallet), <https://deepwiki.com/apache/incubator-teaclave-trustzone-sdk>, 请深度阅读相关文档, 给出一个基于qemu的开发步骤, 假设我们要开发一个rust应用, 提供对用户请求的签名, 而签名服务依赖调用TEE的TA应用

好的, 这是一个非常专业且有价值的技术方向。将签名密钥等核心资产置于 TEE (Trusted Execution Environment) 中是保障区块链、物联网和各类关键应用安全的最佳实践之一。

你提供的链接 `apache/incubator-teaclave-trustzone-sdk` 是核心, 这是一个基于 Rust 为 ARM TrustZone 开发可信应用(TA)的软件开发工具包。它极大地简化了原本非常复杂的 TEE 开发流程。另一个链接是对其的解读。

我将深度研究相关文档, 并为你提供清晰、可操作的第一步指南, 详细说明如何在本地使用 **QEMU** 初始化一个完整的 **TEE** 开发和调试环境, 并基于 `eth_wallet` 范例, 构建一个你自己的签

名服务。

---

## 核心概念解析

在开始之前，我们必须理解几个关键概念：

1. **TEE (Trusted Execution Environment)**: CPU 内的一个安全区域，能保证其中运行的代码和数据的机密性与完整性，即使操作系统本身(如 Linux)也无法访问。ARM TrustZone 就是一种硬件实现。
2. **Normal World vs. Secure World**:
  - **Normal World**: 运行我们熟悉的普通操作系统，如 Linux。你的主应用(Client Application, CA)运行在这里。它是“不可信”的。
  - **Secure World**: 运行 TEE OS(本项目中使用 **OP-TEE**)和可信应用(Trusted Application, TA)。它是“可信”的。
3. **CA (Client Application)**: 运行在 Normal World 的普通 Rust 应用。它负责处理用户请求，但当需要执行敏感操作(如签名)时，它会向 TEE 发起调用。在范例中，对应 `host/` 目录。
4. **TA (Trusted Application)**: 运行在 Secure World 的安全 Rust 应用。它持有私钥，并对外提供一个安全的签名接口。它与外界隔离，只有 CA 能通过严格定义的接口调用它。在范例中，对应 `ta/` 目录。
5. **QEMU**: 一个强大的模拟器。在这里，它将模拟一个支持 ARM TrustZone 的硬件平台，让我们无需购买昂贵的开发板就能在本地进行 TEE 开发和调试。

我们的目标:在 QEMU 模拟的 Normal World Linux 中运行一个 CA, 这个 CA 调用 Secure World 中的 TA 来对一个数据哈希进行签名。

---

## 第一步:本地 QEMU TEE 开发环境初始化与构建

这一步是整个开发流程的基石，也是最耗时的一步。它的目标是编译整个 TEE 运行环境(包括 OP-TEE OS、工具链)并成功在 QEMU 中启动。

假设环境: 强烈推荐使用 **Ubuntu 20.04** 或 **22.04 LTS**。TEE 开发对环境依赖较为严格，使用官方推荐的系统可以避免大量问题。

### 步骤 1.1: 安装系统依赖

首先，安装构建 SDK 和 OP-TEE 所需的所有软件包。

Bash

```
sudo apt-get update
sudo apt-get install -y \
    build-essential \
    git \
    curl \
```



```
python3 \  
python3-pip \  
uuid-dev \  
libssl-dev \  
libffi-dev \  
libglib2.0-dev \  
libpixman-1-dev \  
libsdl2-dev \  
ninja-build \  
iasl \  
ncurses-dev
```

### 步骤 1.2: 克隆 Teaclave TrustZone SDK

这是我们的核心工作区。请务必使用 `--recursive` 参数，因为它需要下载 OP-TEE 等重要的子模块。

Bash

```
git clone --recursive https://github.com/apache/incubator-teaclave-trustzone-sdk.git  
cd incubator-teaclave-trustzone-sdk
```

注意: 如果你忘记了 `--recursive`, 可以进入目录后运行 `git submodule update --init --recursive` 来补救。

### 步骤 1.3: 下载并配置交叉编译工具链

TEE 的 TA 部分运行在 ARM 架构上, 我们需要一个交叉编译器。SDK 已经为我们准备好了下载脚本。

Bash

```
# 这个命令会自动下载并解压所需的 aarch64 和 armv7a 的工具链  
make toolchains
```

此步骤会下载几个较大的文件, 请耐心等待。下载完成后, `toolchains/` 目录下会包含所需的编译器。

### 步骤 1.4: 构建完整的 OP-TEE for QEMU 环境

这是最关键的一步。SDK 的 Makefile 封装了编译 OP-TEE OS、构建 QEMU 镜像等所有复杂操作。我们只需执行一个命令。

重要提示: 首次构建会从头编译所有组件, 根据你的机器性能, 可能需要 **30分钟到数小时**。请确保网络稳定和磁盘空间充足(至少 20GB)。

Bash

```
# 编译支持 aarch64 (64位 ARM) 的 QEMU v8 环境
make optee-qemuv8
```

如果一切顺利，你不会看到任何 `Error` 退出。最终，构建产物会位于 `out/` 目录下。

### 步骤 1.5: 启动并验证 QEMU TEE 环境

构建完成后，我们来启动这个模拟环境，验证一切是否正常。

Bash

```
# 运行刚刚构建的 QEMU v8 环境
make run-qemuv8
```

执行后，你会看到两个 **QEMU** 窗口 弹出：

1. 一个窗口 (标题通常是 **QEMU (SECURE)**): 这是 **Secure World** 的终端。你会看到 OP-TEE 的启动日志，最后会停止在日志输出上。这是正常的，代表 TEE OS 正在运行。
2. 另一个窗口 (标题通常是 **QEMU**): 这是 **Normal World** 的终端。你会看到一个 Linux 启动过程，最后停在 `root@vexpress-qemu_armv8a` 的登录提示符下。

验证:

在 Normal World (`root@...`) 的终端里，我们可以运行 `xtest` 来验证 TEE 功能是否正常。`xtest` 是 OP-TEE 自带的一套功能测试集。

Bash

```
# 在 Normal World 的 QEMU 终端中输入:
xtest -l 3
```

如果你看到大量的 `PASS` 或 `SKIPPED` 测试用例，并且没有 `FAIL`，最后显示一个总结报告，那么恭喜你！你的本地 **TEE** 开发环境已经成功搭建！

按 `Ctrl-A` 然后按 `X` 可以退出 QEMU。

---

## 第二步: 基于范例构建你的签名应用

现在有了可用的 TEE 环境，接下来将基于 `eth_wallet` 范例构建我们自己的应用。

### 步骤 2.1: 理解 `eth_wallet` 范例结构

让我们先剖析这个范例，了解 TEE 应用的构成：

- `projects/web3/eth_wallet/`
  - `host/`: **CA (Client Application)** 的代码。这是一个标准的 Rust `main.rs`，它会加载 TA 并调用其功能。
  - `ta/`: **TA (Trusted Application)** 的代码。这是运行在 TEE 中的安全逻辑，比如生成密钥、签名等。
  - `invocation/`: 调用接口定义。定义了 CA 和 TA 之间如何通信，包括命令 ID (`CommandId`) 和共享的数据结构。这是两者之间的“合约”。
  - `proto/`: Protobuf 定义，用于序列化复杂数据。
  - `Makefile`: 用于构建此范例的脚本。

## 步骤 2.2: 构建并运行 `eth_wallet` 范例

首先，我们完整地构建并运行一次官方范例，以确保构建流程通畅。

1. 构建范例：  
在 SDK 的根目录下，执行：
2. Bash

```
make examples-eth_wallet
```

- 3.
4. 这个命令会：
  - 编译 `ta/` 目录下的 TA 代码，生成一个经过签名的 `xxxx.ta` 文件。
  - 编译 `host/` 目录下的 CA 代码，生成一个可执行文件 `eth_wallet`。
  - 将这两个文件打包到 `out/optee-qemu8-3.15.0/shared_folder` 目录下。这个目录会自动挂载到 QEMU 的 Normal World 中。
5. 在 **QEMU** 中运行范例：
  - 首先，像之前一样启动 QEMU: `make run-qemu8`
  - 在 **Normal World** (`root@...`) 终端中，`shared_folder` 被挂载到了 `/mnt` 目录。我们进入该目录并运行 CA。
6. Bash

```
# 在 QEMU Normal World 终端中输入
```

```
cd /mnt
```

```
ls
```

```
# 你应该能看到 eth_wallet 和一个UUID命名的.ta文件
```

```
# 运行 CA
```

```
./eth_wallet
```

- 7.
8. 你将看到程序的输出，包括生成的公钥和对一条消息的签名。同时，在 **Secure World** 的

终端窗口，你也会看到 TA 被加载和调用的日志。

### 步骤 2.3: 创建并改造你自己的签名应用

现在，我们正式开始开发自己的应用。最简单的方式是复制和修改 `eth_wallet`。

1. 复制项目：
2. Bash

# 在 incubator-teaclave-trustzone-sdk/projects/ 目录下

```
cd projects
```

```
cp -r web3/eth_wallet/ ./my_signer
```

```
cd ../ # 回到 SDK 根目录
```

- 3.
- 4.
5. 修改调用接口 (invocation/):  
打开 `projects/my_signer/invocation/src/lib.rs`。这是你定义服务接口的地方。
6. Rust

```
// 原来的 CommandId
```

```
// pub enum CommandId {
```

```
//   GetPubkey,
```

```
//   Sign,
```

```
// }
```

```
// 修改为你自己的命令
```

```
pub enum CommandId {
```

```
    // 可以保留获取公钥的功能
```

```
    GetPubkey,
```

```
    // 修改 Sign 为更通用的签名接口
```

```
    SignHash,
```

```
}
```

```
// 定义 SignHash 命令需要的数据结构
```

```
#[derive(Serialize, Deserialize, Debug)]
```

```
pub struct SignHashPayload {
```

```
    pub hash: [u8; 32], // 假设我们总是对 32 字节的哈希签名
```

```
}
```

- 7.
- 8.
9. 修改 TA (`ta/main.rs`):  
打开 `projects/my_signer/ta/src/main.rs`。这是实现安全逻辑的地方。
  - 在 `invoke_command_entry_point` 函数中，修改 `match` 分支来处理你的新命令。

## 10. Rust

```
fn invoke_command_entry_point(...) -> optee_tec::Result<()> {
    // ...
    match CommandId::from_u32(cmd_id) {
        Some(CommandId::GetPubkey) => {
            // ... (可以保留)
        }
        Some(CommandId::SignHash) => {
            // 调用新的签名处理函数
            sign_hash(params)
        }
        None => {
            // ...
        }
    }
}

// 实现新的 sign_hash 函数
fn sign_hash(params: &mut [Param]) -> optee_tec::Result<()> {
    // 从 CA 获取输入
    let mut p0 = unsafe { params[0].as_memref().unwrap() };
    let payload = from_slice::<SignHashPayload>(p0.buffer()).unwrap();

    // 模拟从 TEE 内部安全存储中获取私钥
    // 在真实应用中, 此密钥应在首次启动时生成并加密存储
    let private_key = get_my_private_key(); // 你需要自己实现这个函数

    // 使用 Rust 加密库进行签名
    // **重要**: 这里的库必须是 `no_std` 兼容的 !
    // 例如 `k256` 或 `ed25519-dalek`
    use k256::ecdsa::{SigningKey, Signature, signature::Signer};
    let signing_key = SigningKey::from_bytes(&private_key).unwrap();
    let signature: Signature = signing_key.sign(&payload.hash);

    // 将签名结果返回给 CA
    let mut p1 = unsafe { params[1].as_memref().unwrap() };
    // to_vec() 可能需要 allocator, 如果 TEE 中没有, 需要直接写入 buffer
    p1.buffer()[..signature.to_bytes().len()].copy_from_slice(&signature.to_bytes());
    p1.set_updated_size(signature.to_bytes().len());

    Ok(())
}
```

11.

12.

13. 修改 CA (host/main.rs):

打开 projects/my\_signer/host/main.rs。修改它来调用新的 SignHash 命令。

## 14. Rust

```
// ...
// 准备要签名的数据哈希
let data_to_sign = b"this is my request";
let hash = sha256::digest(data_to_sign); // 使用 sha256 crate

// 准备 payload
let payload = SignHashPayload { hash: hash.into() };
let payload_bytes = to_vec(&payload).unwrap();

// 准备 OP-TEE 调用参数
let p0 = Param::new(ParamType::MemrefReadonly, &payload_bytes, 0, 0);
// 准备一个 buffer 来接收签名结果
let mut signature_buffer = [0u8; 64]; // ECDSA P-256 签名是 64 字节
let p1 = Param::new(ParamType::MemrefOutput, &mut signature_buffer, 0, 0);

// 调用 TA 的 SignHash 命令
session.invoke_command(CommandId::SignHash as u32, &mut [p0, p1, Param::none(),
Param::none()]);

println!("Successfully signed the hash!");
println!("Signature: {:?}", &signature_buffer[..p1.updated_size()]);
```

15.

16.

17. 构建和运行你的应用:

- 你需要修改 `projects/my_signer/Makefile`, 将其中所有的 `eth_wallet` 字符串替换为 `my_signer`。
- 在 SDK 根目录, 创建一个新的 make target, 可以参考 `Makefile` 中 `examples-eth_wallet` 的写法。或者简单地, 修改 `Makefile` 中的 `examples-eth_wallet` 目标, 使其指向你的 `my_signer` 目录。
- 运行 `make examples-eth_wallet` (或者你自己的新 target)。
- 启动 QEMU, 进入 `/mnt` 目录, 运行你的新程序 `./my_signer`。

## 总结与展望

通过以上步骤, 你已经完成了从零开始搭建 TEE 开发环境, 并基于一个成熟的范例, 成功构建了一个自定义的可信签名应用的整个流程。

关键点回顾:

- 环境是基础: 使用推荐的 Ubuntu 版本, 耐心等待首次构建。
- 范例是捷径: 不要从零开始, 复制并改造现有范例是最快的方式。
- 接口是合约: `invocation` 目录是 TA 和 CA 之间的桥梁, 修改功能要先从这里开始。

- `no_std` 是约束: TA 中使用的任何 Rust Crate 都必须支持 `no_std` 环境。

接下来, 你可以继续深化开发, 例如:

- 实现一个真正的密钥生成和持久化机制(使用 OP-TEE 的安全存储 API)。
- 集成更复杂的加密算法。
- 将 CA 从一个简单的命令行工具, 改为一个响应网络请求的 Web 服务(如使用 Axum 或 Actix-web), 对外提供签名 API。

这个本地 QEMU 环境为你提供了一个功能完备、可调试的沙箱, 是你通往 TEE 安全开发的坚实第一步。