LPC22

Kernel TEE subsystem evolution

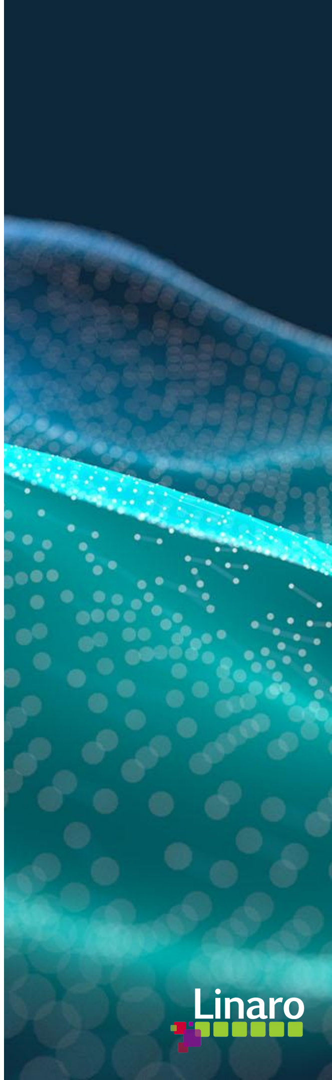
Sumit Garg
Senior Engineer, Linaro Ltd.

Who am I?

- Member of Linaro SSE team

- Have a keen interest in platform security
  - Linux kernel: TEE subsystem reviewer, maintainer for TEE trusted keys and OP-TEE hwrng.
  - OP-TEE and TF-A: platform maintainer.

- Other areas of interest being toolchain and debugging
  - meta-arm-toolchain layer maintainer
  - ftrace for OP-TEE
  - Contributions to kgdb



What is TEE?

Trusted Execution Environment

Isolated execution environment running alongside Rich Execution Environment (REE) like Linux OS.

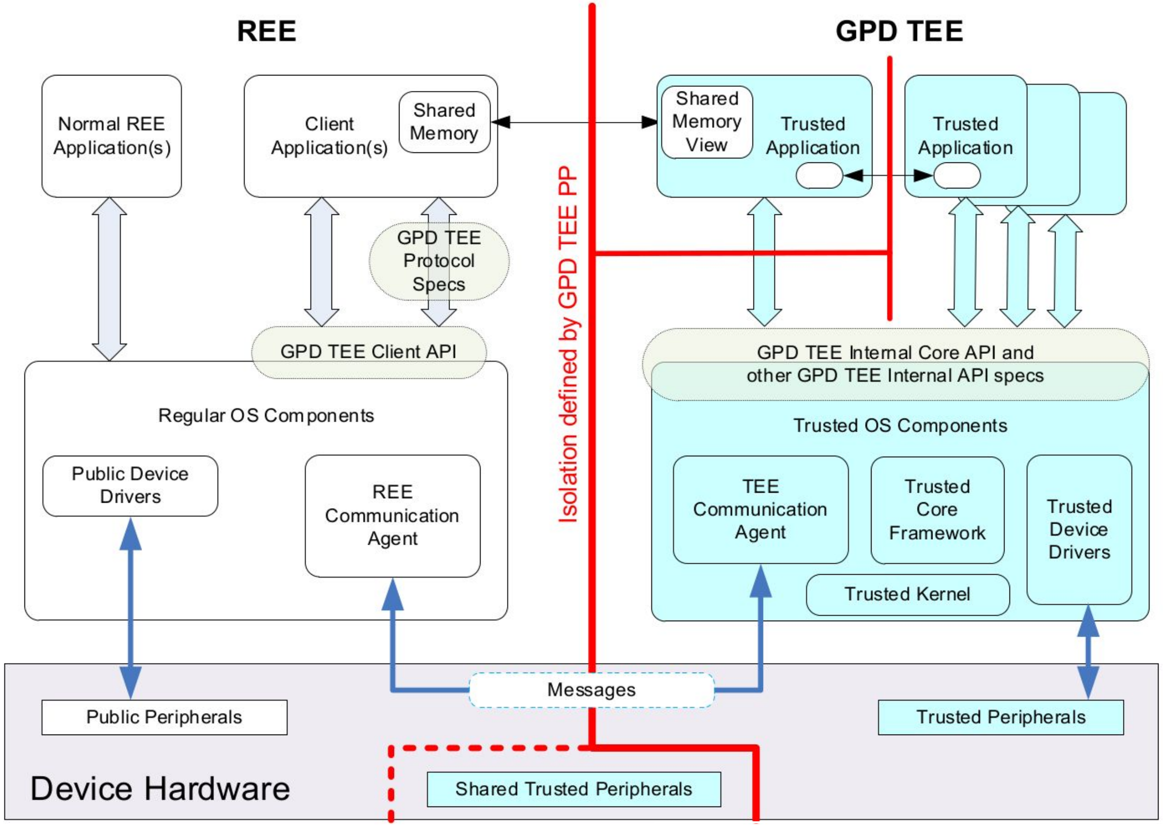
Provides capability to isolate trusted code and corresponding resources like memory, devices, etc.

Isolation is backed by hardware security features such as Arm TrustZone, AMD Secure Processor, etc.

Industry standard: GlobalPlatform Defined TEE (specs)

Generic TEE Software Architecture

Figure 3-1: TEE Software Architecture



Source: GPD_TEE_SystemArchitecture_v1.3_PublicRelease

Open Portable TEE (OP-TEE)

An open source TEE implementation with **mainline** Linux kernel driver

Supported architectures: arm, arm64 and risc-v (early stages)

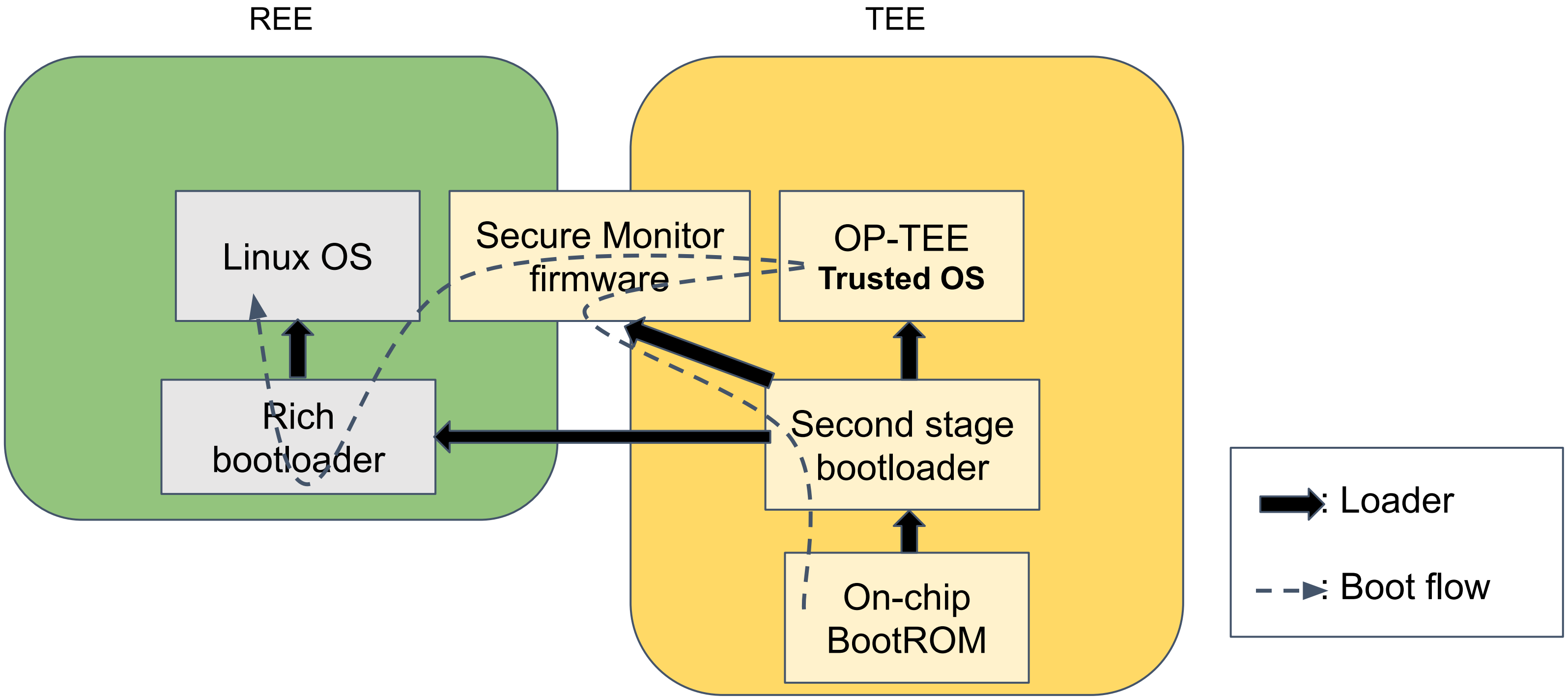
Design goals:

- Isolation
- Small footprint
- Portability

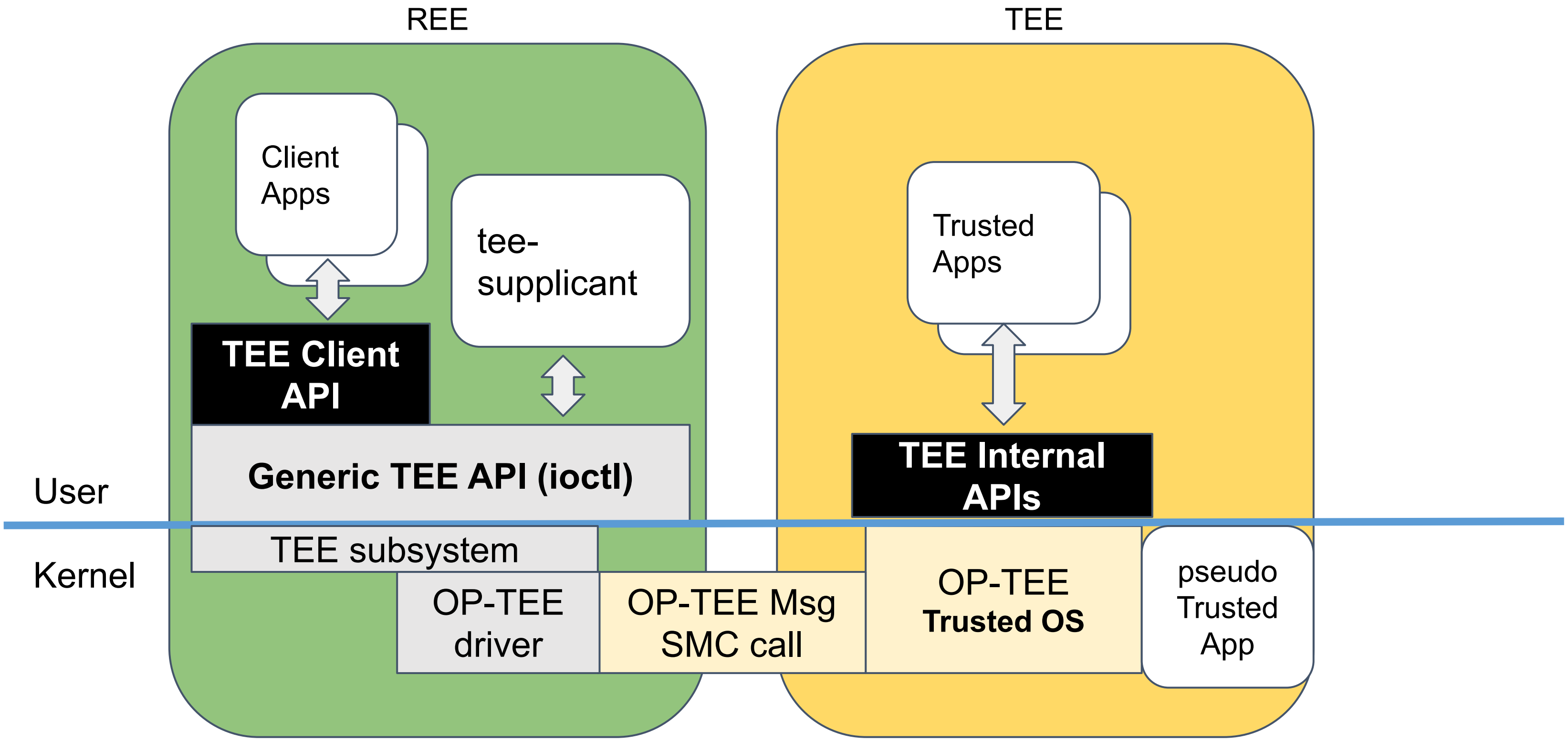
You don’t need to have actual hardware to get hands-on experience with OP-TEE:

- Use Qemu, steps here.

OP-TEE: boot


REE
TEE
Linux OS
Secure Monitor firmware
OP-TEE
Trusted OS
Rich bootloader
Second stage bootloader
On-chip BootROM
: Loader
: Boot flow

OP-TEE: runtime



REE

TEE

Client Apps

tee-supplicant

TEE Client API

Generic TEE API (ioctl)

User

Kernel

TEE subsystem

OP-TEE driver

OP-TEE Msg SMC call

Trusted Apps

TEE Internal APIs

OP-TEE **Trusted OS**

pseudo Trusted App

Kernel TEE subsystem

Kernel TEE subsystem

Introduced in Linux kernel version 4.12

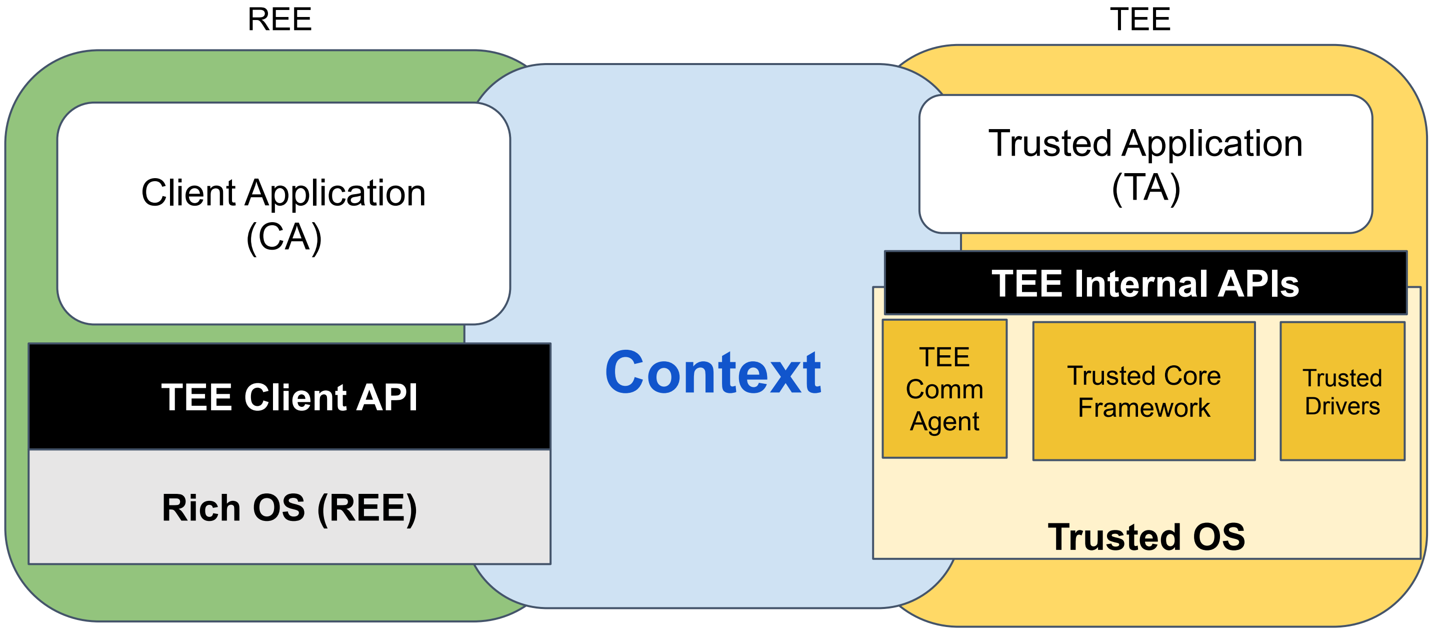
TEE subsystem

- Registration/unregistration of TEE drivers
- Shared memory between normal world and secure world
- User-space TEE IOCTL interface

OP-TEE driver

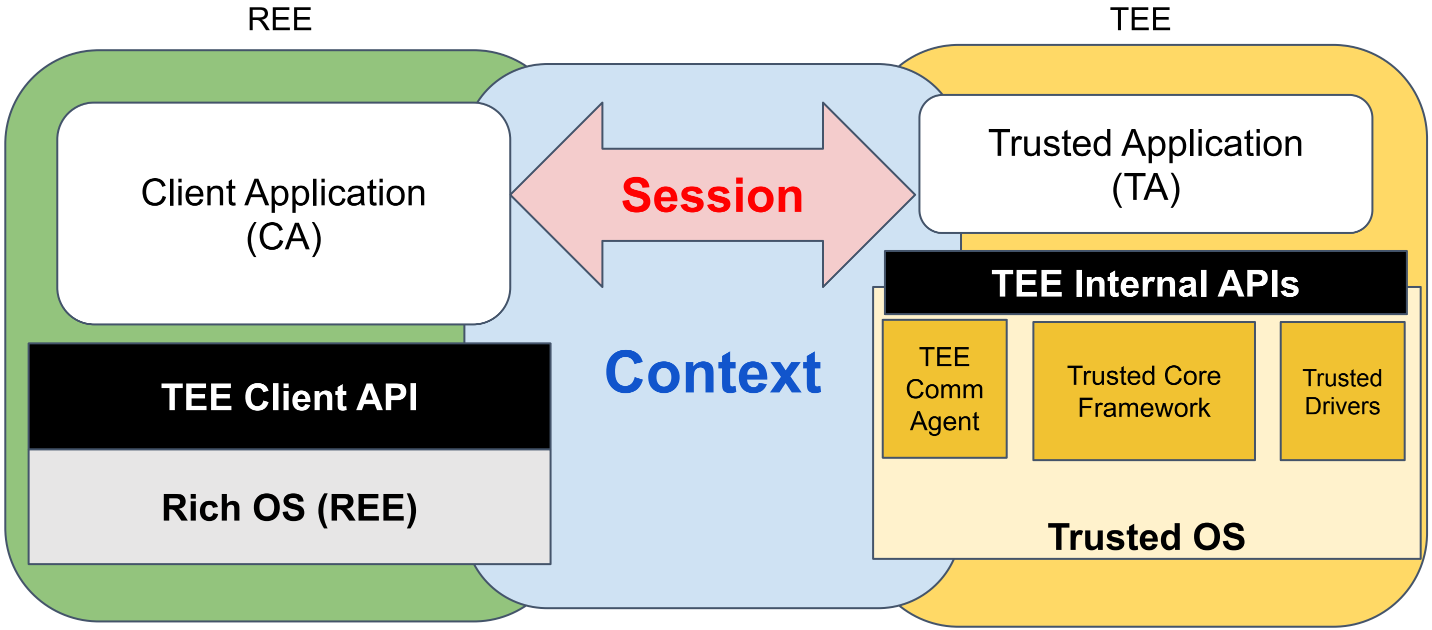
- Targets ARM and ARM64
- Shared memory support via reserved memory
- TEE supplicant support

TEE communication API



TEEC_{Initialize/Finalize}Context()

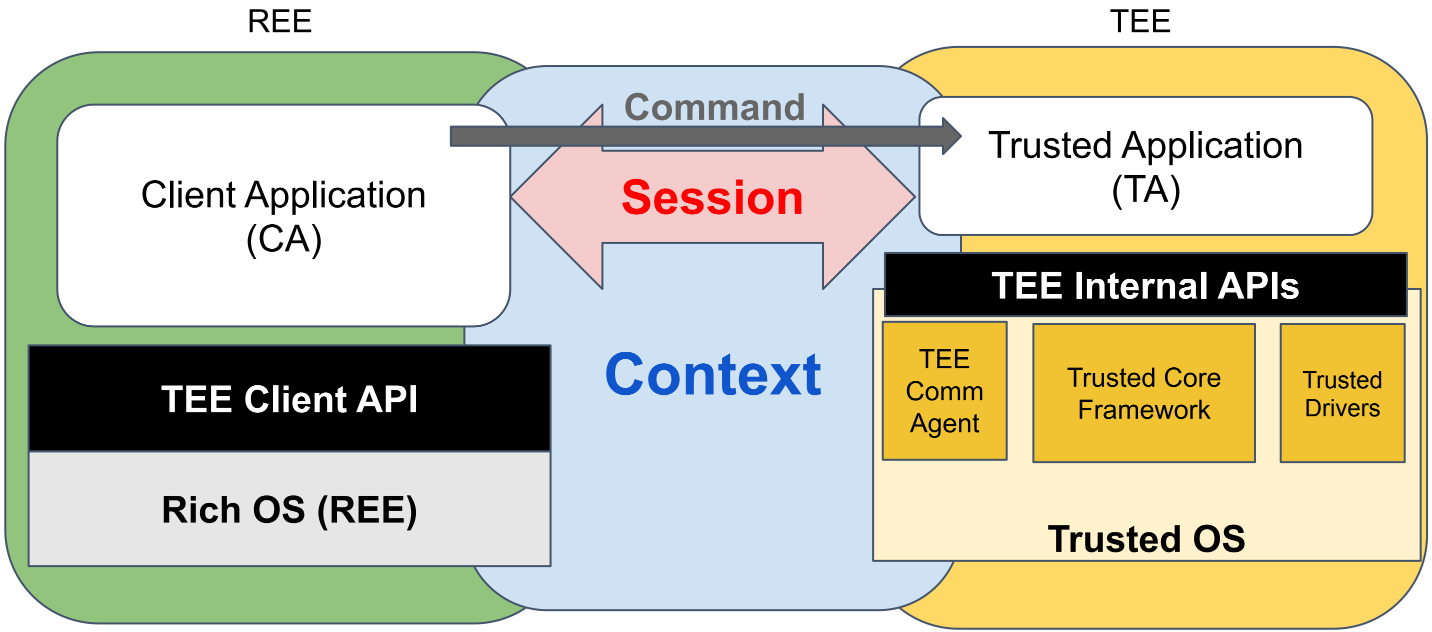
TEE communication API



TEEC_{Initialize/Finalize}Context() TEEC_{Open/Close}Session()

Linaro

TEE communication API



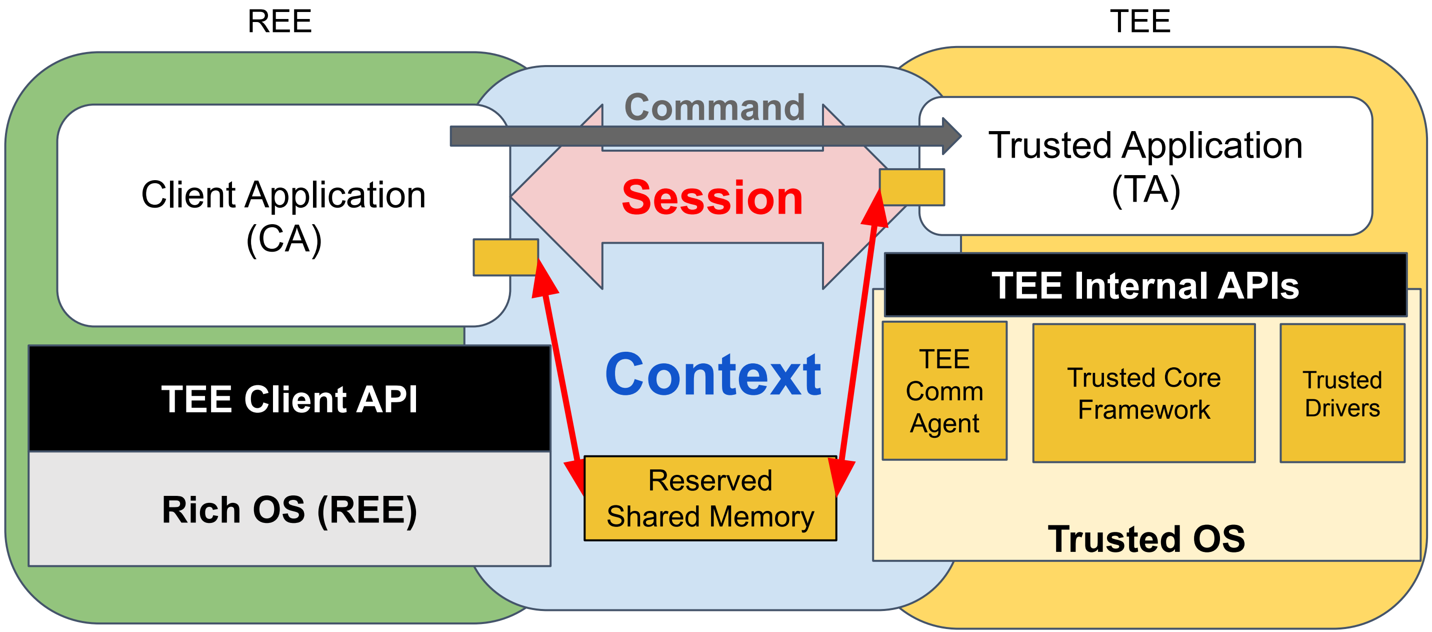
TEEC_{Initialize/Finalize}Context()

TEEC_{Open/Close}Session()

TEEC_InvokeCommand()

Linaro

TEE communication API



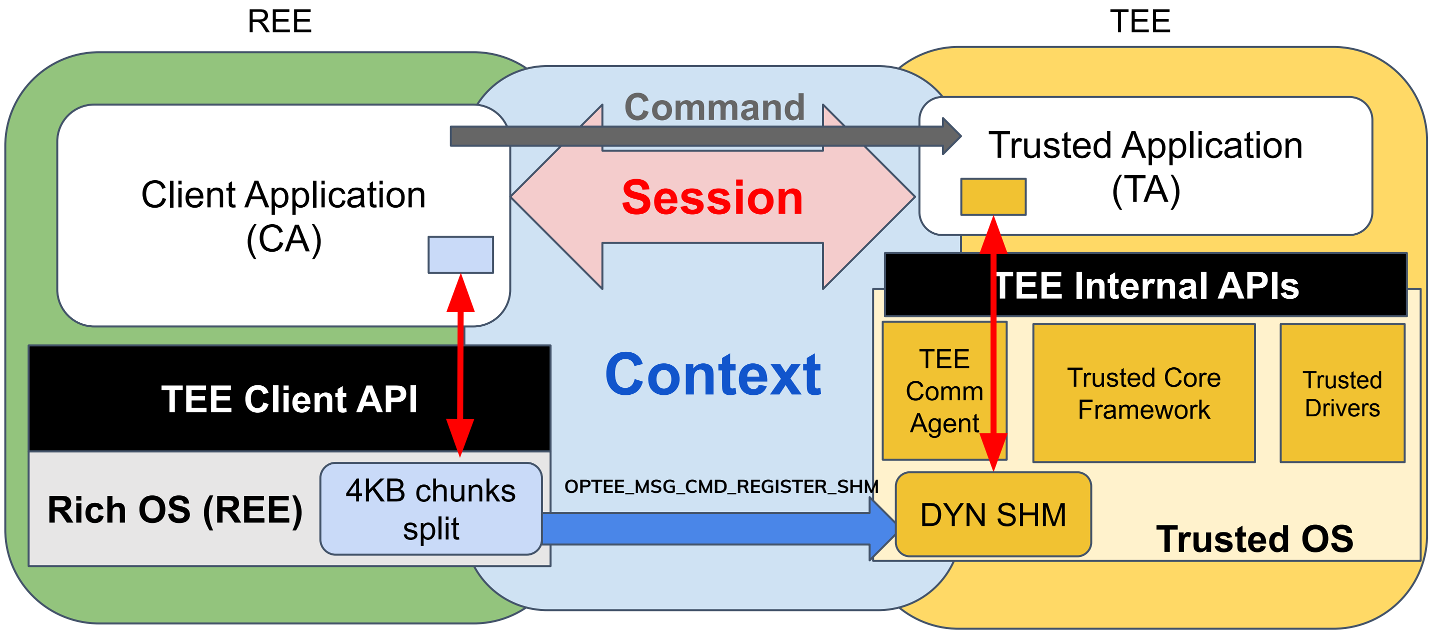
TEEC_{Initialize/Finalize}Context()

TEEC_{Open/Close}Session()

TEEC_InvokeCommand()

TEEC_{Allocate/Register/Free}SharedMemory()

Evolution: Dynamic shared memory support



TEEC_{Initialize/Finalize}Context()

TEEC_{Open/Close}Session()

TEEC_InvokeCommand()

TEEC_{Allocate/Register/Free}SharedMemory()

Evolution: In-kernel API

Linux kernel provides an **internal** TEE client interface for kernel to interact with TEE.

Allows TAs to be the **hardware** backend for existing kernel subsystems (rng, crypto, secure storage, etc).

Exported major APIs:

- tee_client_open_context()
- tee_client_close_context()
- tee_client_get_version()
- tee_client_open_session()
- tee_client_close_session()
- tee_client_invoke_func()
- tee_shm_alloc_kernel_buf()
- tee_shm_register_kernel_buf()
- tee_shm_free()

Evolution: TEE bus framework

Allows kernel to **automatically** react to the TAs provided by TEE in order to load and probe drivers corresponding to these TAs.

The TAs are identified via Universally Unique Identifier (**UUID**). The drivers need to register a **table** of supported TA UUIDs.

TEE bus framework registers following APIs:

- **match()**: iterates over the driver UUID table to find a corresponding match for TA UUID.
- **uevent()**: notifies user-space (udev) whenever a new TA is registered on TEE bus for auto-loading of modularized drivers.

OP-TEE enumeration support

Linux kernel TEE bus framework allows for device enumeration to be **implementation specific** like for OP-TEE etc.

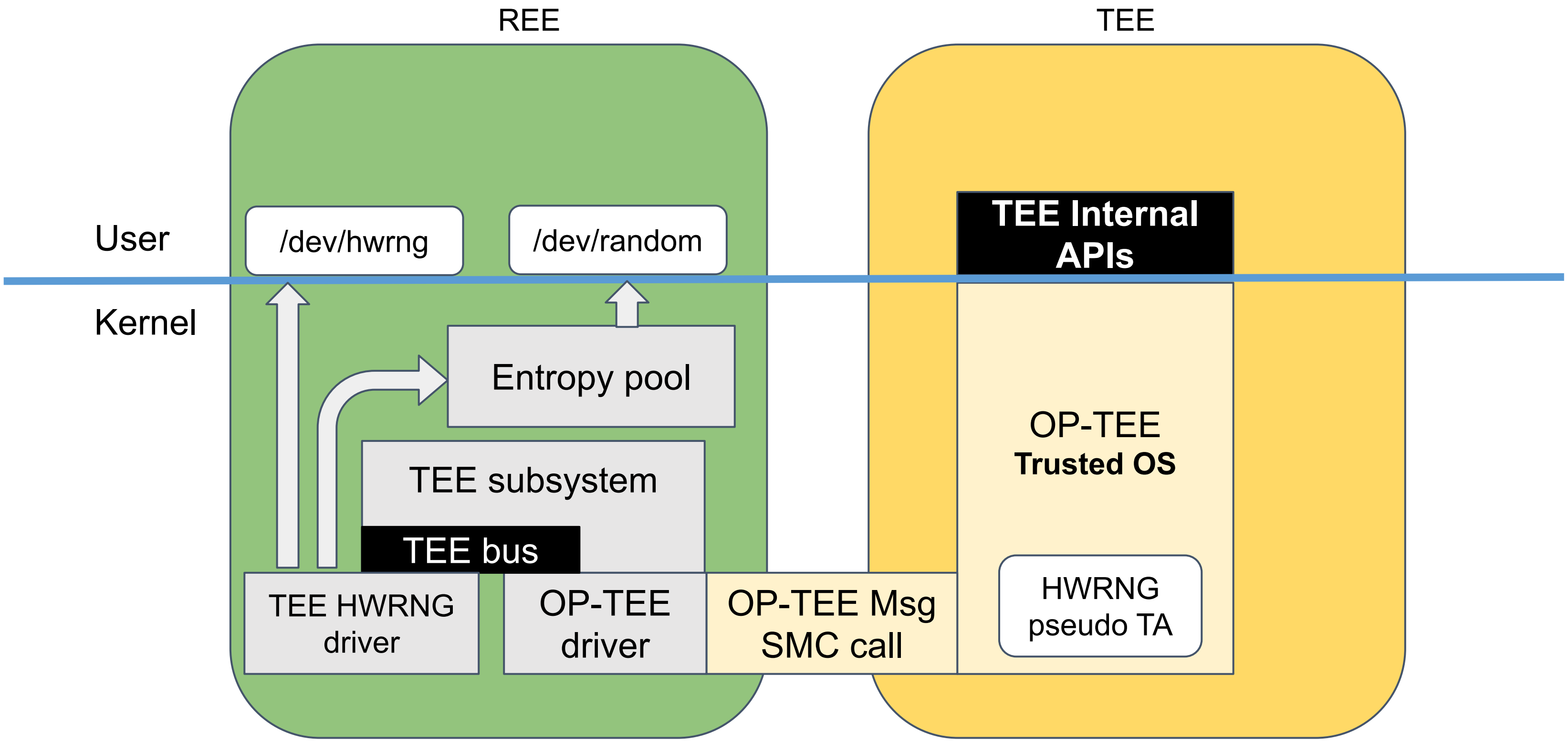
OP-TEE provides a pseudo TA to enumerate TAs which can act as devices/services for TEE bus.

To enumerate devices **a device pseudo TA function** is invoked to fetch an array of device UUIDs, which are then registered with TEE bus as "**optee-ta-<UUID>**" devices.

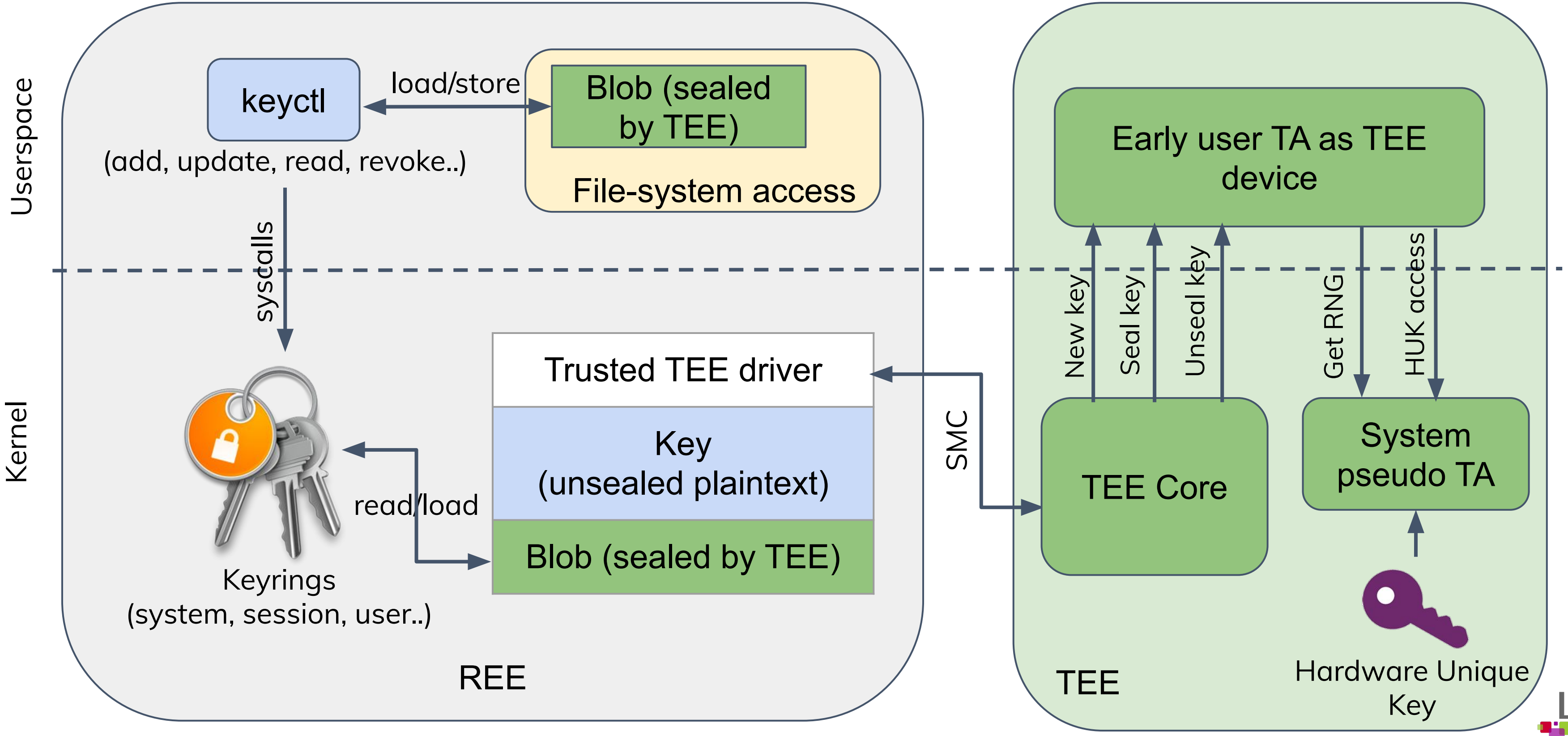
Currently OP-TEE allows **pseudo/early TAs** to be enumerated as TEE bus devices/services.

Use-cases

Use-cases: HWRNG



Use-cases: Trusted Keys



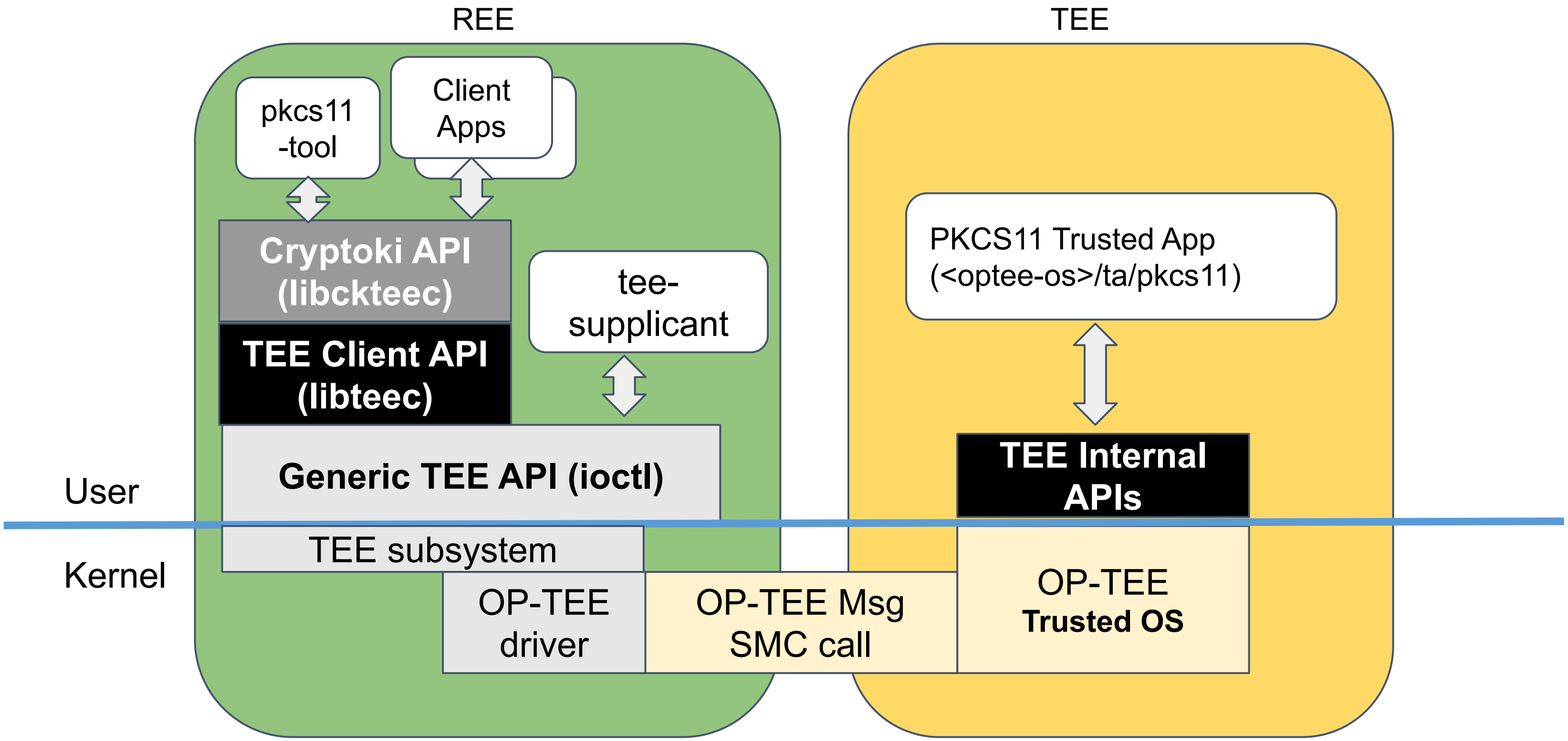
Use-cases: firmware TPM

A firmware TPM with regards to OP-TEE is implemented as a trusted application. It works transparently with the existing TSS stack as compatible with a regular TPM chip as possible.

Microsoft's fTPM

- TCG TPM 2.0 spec has lots of code written in C (main contributor is Microsoft).
- Microsoft took that code and created fTPM.
- Runs on Arm (TrustZone) using OP-TEE as a reference implementation.
- Linux kernel fTPM driver
  - `drivers/char/tpm/tpm_ftpm_tee.c`

Use-cases: PKCS#11 Token



Future

Kernel runtime measurements

A step towards **tamper proof** runtime environment for **trustworthy** systems.

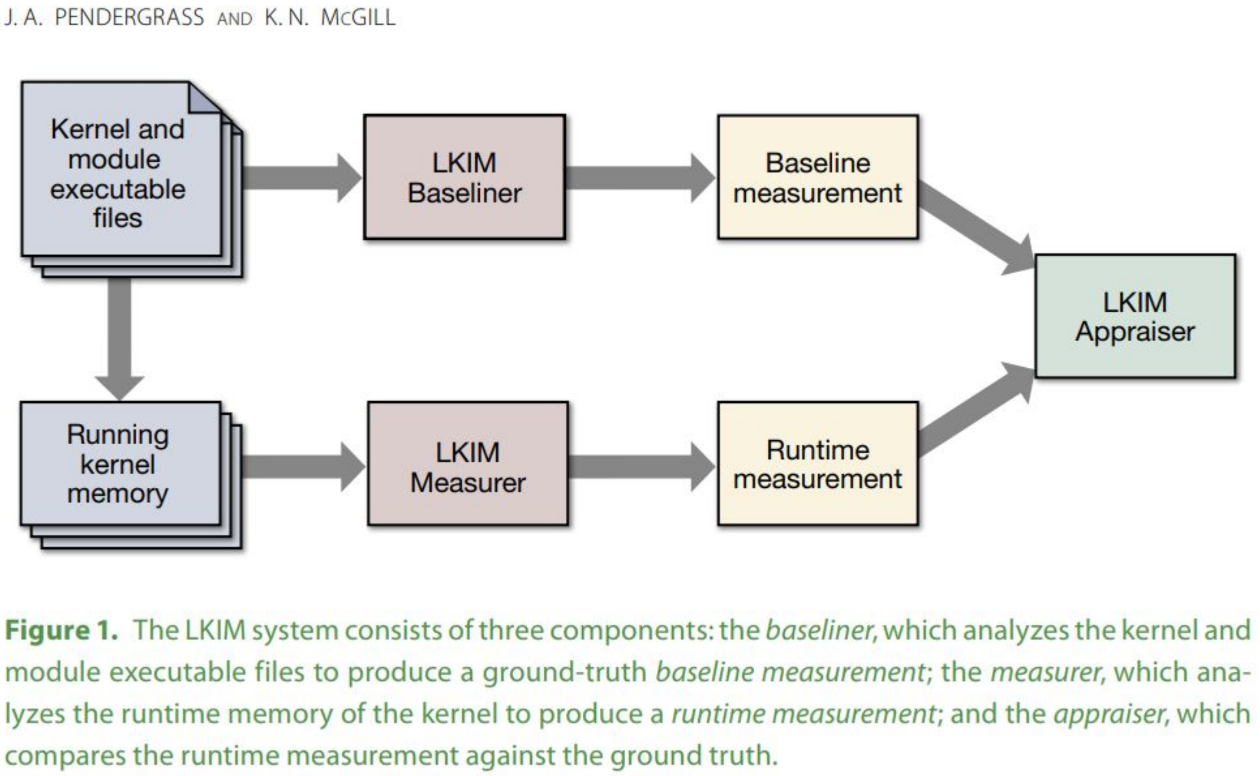
Allows **periodic** trust building to assure that a piece of **executing** kernel is behaving consistently with its static definition.

Makes it more **difficult** to install “kernel-level rootkits”.

Some prior research papers:

- LKIM: The Linux Kernel Integrity Measurer
- KIMS: Kernel Integrity Measuring System based on TrustZone

Prior Art: The Linux Kernel Integrity Measurer



J. A. PENDERGRASS AND K. N. McGILL

Figure 1. The LKIM system consists of three components: the *baseliner*, which analyzes the kernel and module executable files to produce a ground-truth *baseline measurement*; the *measurer*, which analyzes the runtime memory of the kernel to produce a *runtime measurement*; and the *appraiser*, which compares the runtime measurement against the ground truth.

Source: LKIM: The Linux Kernel Integrity Measurer

Kernel runtime measurements: TEE

TEE can act as a Dynamic Root of Trust for Measurement (**DRTM**).

OP-TEE already has support for runtime measurements that can be **attested** remotely.

- Trusted OS memory
- TAs memory

Similar service can be exposed via pseudo TA where:

- Private attestation key is **only** accessible to secure world.
- Linux kernel memory can be registered **dynamically** for measurements.
- Replay attacks are prevented by the use of a **nonce** in the authentication request.

Which kernel portions should we measure?

- Self
  - The driver responsible for communication with TEE
- Kernel text section
- Kernel rodata section
- Modules specific text section
- Modules specific rodata section
- Critical data structures
  - Security subsystem related
  - Modules related
- Arch specific code/data section
- and others

Kernel runtime measurements: proposal 1

TEE interface exposed via securityfs “/sys/kernel/security/tee/”

- Kernel TEE communication agent has already registered a **static list** of kernel memory ranges with TEE.
- Update nonce and trigger new measurement:

```
$ echo {nonce} > /sys/kernel/security/tee/measure
```

- Fetch measurement:

```
$ cat /sys/kernel/security/tee/measurement
{start symbol}:{end symbol}:{hash}:{sign(hash | nonce)}
...
```

- Fetch attestation public key (once during device provisioning):

```
$ cat /sys/kernel/security/tee/attestation_pub_key
{algo}:{exp}:{modulus}
```

Which kernel portions should we measure?

- Self
    - The driver responsible for communication with TEE
- Kernel text section
- Kernel rodata section
- Modules specific text section
- Modules specific rodata section
- Critical data structures
    - Security subsystem related
    - Modules related
- Arch specific code/data section
- **Random measurements**
- and others

Which kernel portions should we measure?

- Self
  - The driver responsible for communication with TEE
- Kernel text section
- Kernel rodata section
- Modules specific text section
- Modules specific rodata section
- Critical data structures
  - Security subsystem related
  - Modules related
- Arch specific code/data section
- **Random measurements**
- and others

In order to serve random measurements, the interface can be made flexible to allow the attester app to tell which kernel portions to measure.

So how about the attester app providing the kernel memory range via symbols to be measured?

- **System.map** can be useful

Kernel runtime measurements: proposal 2

TEE interface exposed via securityfs “/sys/kernel/security/tee/”

- Provide kernel start and end symbols:

```
$ echo {start symbol}:{end symbol} > /sys/kernel/security/tee/measure_range
```

- Update nonce and trigger new measurement:

```
$ echo {nonce} > /sys/kernel/security/tee/measure
```

- Fetch measurement:

```
$ cat /sys/kernel/security/tee/measurement
{start symbol}:{end symbol}:{hash}:{sign(hash | nonce)}
...
```

- Fetch attestation public key (once during device provisioning):

```
$ cat /sys/kernel/security/tee/attestation_pub_key
{algo}:{exp}:{modulus}
```

Proposal: Threat model

Attack: Man-In-The-Middle (MITM)

Mitigation: Attestation key is secured by TEE

Attack: Replay attack

Mitigation: Nonce, provided in the authentication request, always lead to a new signature.

Attack: Kernel TEE communication agent is compromised, can create duplicate memory regions for measurement

Mitigations:

- Kernel TEE communication agent attestation.
- Random kernel memory measurements.

Thoughts and future work...

- Let’s champion open source TEE frameworks
    - Reduces effort to maintain downstream TEE solutions

- Kernel runtime measurements
    - Plan to send an RFC for initial review

- Open to discuss other interesting TEE use-cases

Thank you

Go to www.linaro.org