Linaro Connect
MADRID 2024 | MAY 14-17 2024

OP-TEE device drivers

Etienne Carrière, ST

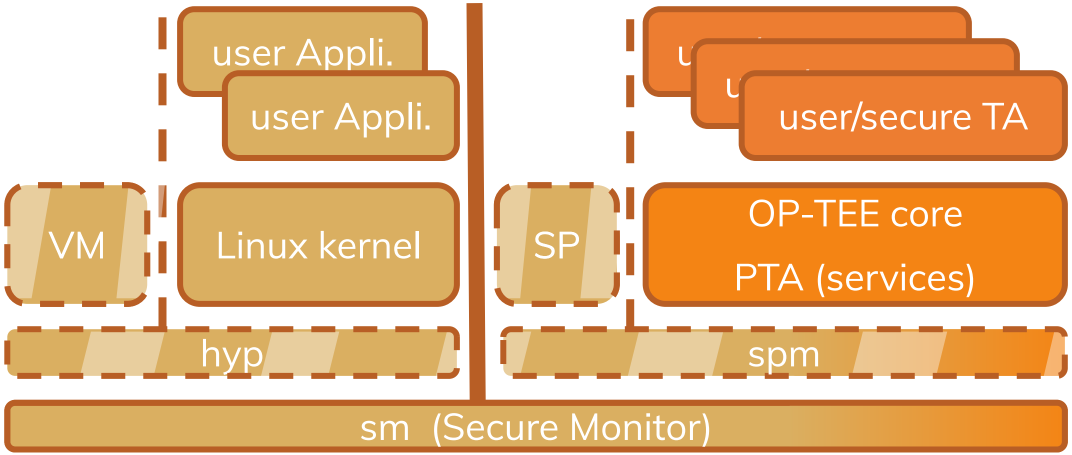
Agenda

- Why device drivers in OP-TEE?
- OP-TEE crypto device integration
- OP-TEE initcalls and probing with DT
- OP-TEE driver frameworks
- What’s next

Drivers in OP-TEE

optional

TEE is about, secrets,
reliable cryptographic operations,
reliable persistent storage,
reliable secure time, …



OP-TEE drivers likely for random number generation and device and platform keys;
possibly for crypto operations;
possibly for resource management (clocks, regulators, busses);
possibly for power management & platform events;
not for persistent storage (no existing OP-TEE drivers, see REE/RPMB FS'es)

OP-TEE crypto device integration

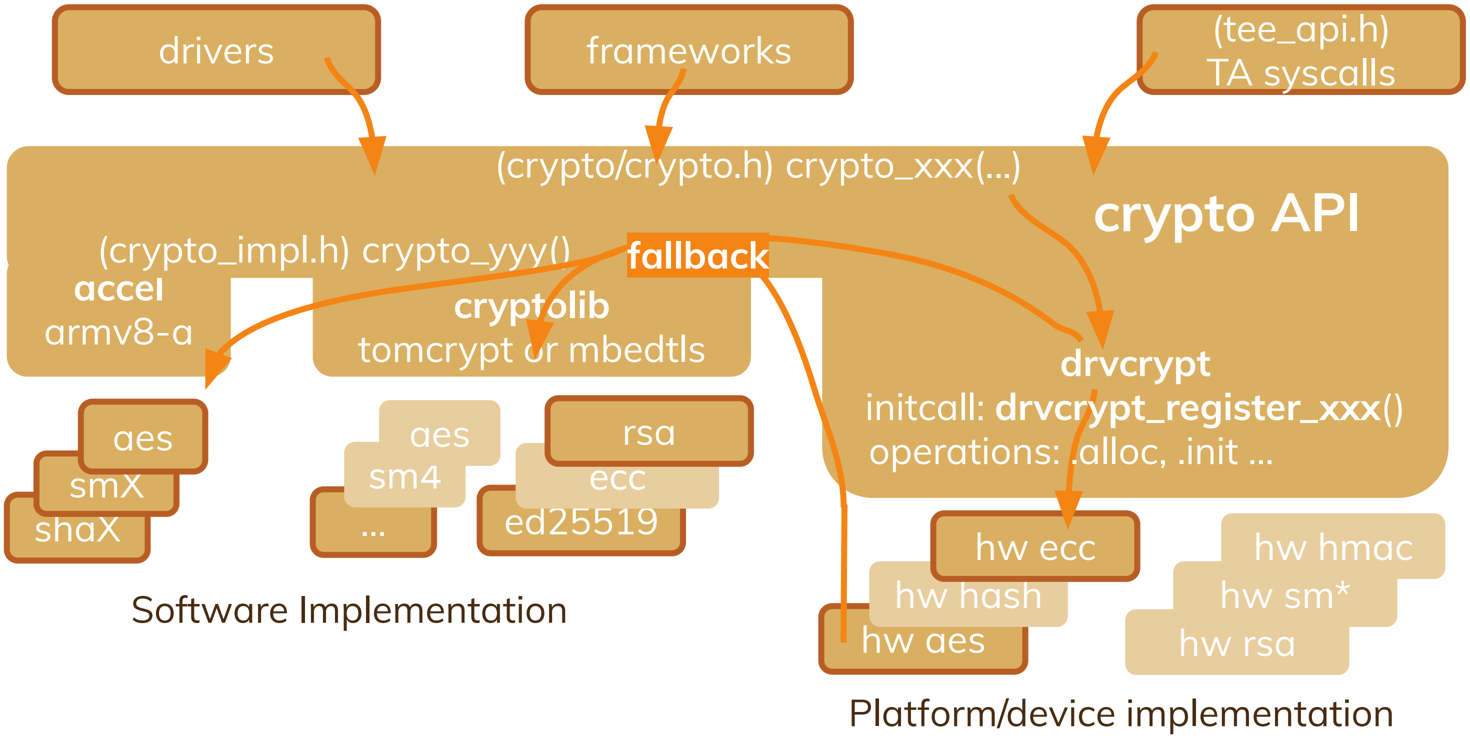
OP-TEE today's latest tag (4.2.0), example: symmetric authenticated encryption (AE)

```
#include <drvcrypt.h>
#include <drvcrypt_authenc.h>

static struct drvcrypt_authenc mydriver_authenc = {
	.alloc_ctx mydriver_ae_allocate,
	.free_ctx = mydriver_ae_free,
	.init = mydriver_ae_initialize,
	.update_aad = mydriver_ae_update_aad,
	.update_payload = mydriver_ae_update_payload,
	.enc_final = mydriver_ae_enc_final,
	.dec_final = mydriver_ae_dec_final,
	.final = mydriver_ae_final,
	.copy_state = mydriver_ae_copy_state,
};

TEE_Result mydriver_register_authenc(void)
{
	return drvcrypt_register_authenc(&mydriver_authenc);
}
```

Other crypto operations:

```
drvcrypt_cipher.h drvcrypt_mac.h
drvcrypt_hash.h drvcrypt_acipher.h
drvcrypt_math.h drvcrypt_asn1_oid.h
```

OP-TEE crypto device integration



OP-TEE initcalls

- init initcalls
  init level 1 : early_init - - - - - - - - platform initialization
  init level 2 : early_init_late - - - - - platform early drivers
  init level 3 : service_init - - - - - - - crypto framework init (incl. RNG)
  init level 4 : service_init_late - - - service with crypto (storage, ...), platform's
  init level 5 : driver_init - - - - - - - - most drivers initialization loop
  init level 6 : driver_init_late - - - - non-secure service setup
  init level 7 : relese_init_resource - only releasing resources (mainly heap)

- 3 preinit levels : non-secure world virtualization, pager mode

- 7 final levels: called before init levels initcalls when `CFG_NS_VIRTUALIZATION=y`
  called before reaching non-secure world

OP-TEE initcalls

```
#include <initcall.h>

static TEE_Result my_early_function(void)
{
	do_my_early_initialization();
	return TEE_SUCCESS;
}

early_init(my_early_function);
```

my_early_function() is called at init level 1, before crypto initializations

```
static TEE_Result my_init_function(void)
{
	do_my_initialization();
	return TEE_SUCCESS;
}

driver_init(my_init_function);
```

my_init_function() is called at init level 4, after crypto initializations

OP-TEE initcalls & DT-drivers

- init initcalls

  init level 1 : early_init - - - - - - - - - platform initialization, clock early probe
  init level 2 : early_init_late - - - - - - platform early drivers, 1st dt_driver probe loop
  init level 3 : service_init - - - - - - - - crypto framework init (incl. RNG)
  init level 4 : service_init_late - - - service with crypto (storage, …), platform's
  init level 5 : driver_init - - - - - - - - most drivers init., 2nd dt_driver probe loop
  init level 6 : driver_init_late - - - - non-secure service setup
  init level 7 : relese_init_resource - only releasing resources (mainly heap)

- When **`CFG_NS_VIRTUALIZATION=y`**: boot_final() is called before init levels
  (same for the nexus final levels initcallls)

- **`CFG_DRIVERS_CLK_EARLY_PROBE=y`**: clock drivers are probed at init level 1 (early_init)

- Early console with serial UART in DT drivers: called before any initcalls

OP-TEE DT-drivers

- Device Tree to describe device interfaces, configurations and dependencies

```
#include <drivers/clk_dt.h>
#include <drvcrypt_authenc.h>
#include <dt_drivers.h>

TEE_Result my_driver_probe(const void *fdt, int node, const void *compat)
{
    ...
    res = clk_dt_get_by_name(fdt, node, “kernel”, &clk);
    ...
    res = drvcrypt_register_authenc(...);
    ...
}

static const struct dt_device_match my_match_table[] = {
      { .compatible = "me,my-driver" },
      { }
};

DEFINE_DT_DRIVER(my_driver) = {
      .name = “my-driver”,
      .match_table = my_match_table,
      .probe = my_driver_probe,
};
```

Probe function can be called at init level 2, before crypto, and at init level 4, after crypto. (TEE_ERROR_DEFER_DRIVER_INIT)

Probe function is called on matching compatible string

OP-TEE DT-drivers

- The dt-driver loop
  - List of nodes for which a driver is to be probed
  - A driver can add new nodes to the list
  - Stop when unresolved dependencies while no new drivers to probe
  - Loop is processed twice: before (init level 2) and after (init level 4) crypto initcall.
- Provider driver registers callback + memref cookie to bind driver handle to DT ref
  dt_driver_register_provider(fdt, node, get_device_callback, memref, driver_type)
- Initial node list: `CFG_DRIVERS_DT_RECURSIVE_PROBE=y|n`
- Panics on probe failure: pending deferred, failing drivers
- Allocated lists (provider drivers, failing driver, etc…) are freed at initcall init level 7
- Perf: many parent node look up to avoid: dozens of milliseconds of boot time

OP-TEE DT-drivers

- DT property "`clocks`" ⟶ clk_dt_get_by_index()
  DT property "`clock-names`" ⟶ clk_dt_get_by_name()
- DT property "*`xxx`*`-gpios`" ⟶ gpio_dt_get_by_index() (release GPIO ⟶ gpio_put())
- I2C bus consumer (child node of an I2C bus node) ⟶ i2c_dt_get_dev()
- DT property "`nvmem-cells`" ⟶ nvmem_get_cell_by_index()
  DT property "`nvmem-cell-names`" ⟶ nvmem_get_cell_by_name()
- DT property "`pinctrl-`*N*" ⟶ pinctrl_get_state_by_idx()
  DT property "`pinctrl-names`" ⟶ pinctrl_get_state_by_name()
- DT property "*`xxx`*`-supply`" ⟶ regulator_dt_get_supply()
- DT property "`resets`" ⟶ rstctrl_dt_get_by_index()
  DT property "`reset-names`" ⟶ rstctrl_dt_get_by_name()
- DT properties "`interrupts`" & "`interrupts-extended`" ⟶ interrupt_dt_get_by_index()
  DT property "`interrupt-names`" ⟶ interrupt_dt_get_by_name()

OP-TEE driver frameworks

- Crypto devices: crypto_xxx() API functions and drvcrypt drivers
  - dt_driver_get_crypto() returns TEE_SUCCESS or TEE_ERROR_DEFER_DRIVER_INIT
- RNG devices: crypto_rng_read()
  - PRNG or HW assisted with hw_get_random_bytes()
  - Should be ready after service_init initcall level, when crypto API is ready
- HUK services
  - tee_otp_get_hw_unique_key()
  - huk_subkey_derive()
- System resources
  - With or without DT for devices description
  - Clocks (clk_enable() …), voltage regulators (regulator_enable() …), GPIOs (gpio_get_value() …), reset controllers (rstctrl_assert() …), interrupt controllers (interrupt_mask() …), OTP cells (nvmem_cell_read() …), pin muxing (pinctrl_apply_state() …), basic I2C bus r/w

What’s next

- STM32 firewall framework (plat-stm32mp1, plat-stm32mp2)
  - Configuring secure hardening using a DT description (P-R #6816)
- Secure interrupts framework
  - Multiplex non-secure events on secure interrupt controllers
  - Expose wake-up services to secure and non-secure worlds (PM)
- plat-stm32mp2 unified image and external DTB
  - Relaxed PTAs enumeration (HWRNG, RTC, Watchdog SMC)
  - Configure SCMI services using a DT description
- Atomic clock gating
- A unified device driver model?

Thank you

Hidden slides

OP-TEE initialization

Hidden slide

Case without non-secure world virtualization (CFG_NS_VIRTUALIZATION=n)

01. Entry Cortex-A core initialization (MMU, GIC, VFP/Neon/SDE, …)
02. Early boot parameters (boot args & transfer list), execution context, pager engine
03. External DT parsing (DRAM, console, TPM log buffer, update non-secure DTB)
04. TA RAM memory pool
05. Preinit initcalls (levels 1 to 3)
06. Init initcalls (levels 1 to 7)
07. Boot final initcall
08. First entry in non-secure world

Hidden slide

OP-TEE DT-drivers

Registering to dt-driver: a clock

```
#include <clk_dt.h>

static TEE_result my_clock_enable(struct clk *clk) { … }

static const struct clk_ops my_clk_ops = { .enable = my_clock_enable };

static TEE_Result dt_get_my_clk(..., void *priv, struct clk **out)
{
    *out = priv_to_clk(priv, ...);
     return TEE_SUCCESS
}

static TEE_Result my_clk_probe(...)
{
    return clk_dt_register(fdt, node, dt_get_my_clk, &my_clk_data);
}

DEFINE_DT_DRIVER(my_clk) = {
     .name = “my-clk”, .probe = my_clk_probe, .match_table = my_match_table,
};
```