Firmware Configuration Framework in TF-A and Chain of Trust

Madhukar Pappireddy, Austin

Manish Badarkhe, Cambridge

March 2021

Introduction to fconf

- Fconf: Firmware Configuration Framework
- An abstraction layer for accessing platform specific information(i.e., property)
- Ex: Info about UART, GIC, IO Policies etc.
- Access data by querying for a “property”: get_value(property).
- Streamline how properties are accessed:
  - Either platform specific information
  - Implementation defined data

Introduction to fconf continued

- It allows requesting entity (BLx image) to query for a property without knowing what kind of backing store is used to hold the property:

1. C structure

```
typedef struct uart_serial_config_t {
        uintptr_t uart_base;
        uint32_t  uart_clk;
}

const uart_serial_config_t uart0_config = {
        .uart_base = 0x1c090000,
        .uart_clk  = 24000000;
}
```

2. Macro definition

```
#define UART0_BASE      0x1c090000
#define UART0_CLK       24000000
```

3. Configuration file(such as dts)

```
v2m_serial0: uart@90000 {
        compatible = "arm,pl011", "arm,primecell";
        reg = <0x090000 0x1000>;
        interrupts = <0 5 4>;
        clocks = <&v2m_clk24mhz>, <&v2m_clk24mhz>;
        clock-names = "uartclk", "apb_pclk";
};
```

- Popular source : Device tree source(dts)

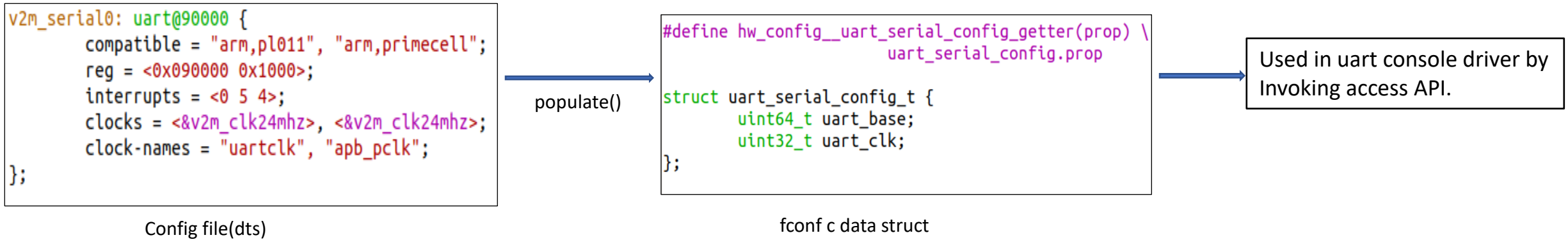
Primary motivation behind fconf

- Reduce the source code fragmentation within family of SoCs due to various platform specific definitions.
- Evaluate the feasibility of having common TF-A BL images across multiple compatible platforms.
- Documentation

https://trustedfirmware-a.readthedocs.io/en/latest/components/fconf/index.html

Design of fconf

- Properties are stored in C data structure: Backward compatible
- Data structures filled with appropriate values once by populate() function
- Platform developer registers the populate() callbacks with fconf statically(build-time)
- Common BLx code calls fconf helper to invoke every populate() function
- Any BLx driver/software can query the property using fconf access API



fconf interfaces

- Step 0: (Build/compile time)
  - Register each populate() callback with the framework
  - Ex: FCONF_REGISTER_POPULATOR(HW_CONFIG, uart_config, fconf_populate_uart_config)
- Step 1: (Optional pre-requisite if source is dtb):
  - Load all necessary configuration files(dtb) into memory
  - Ex: fconf_load_config(hw_config)
- Step 2: fconf iterates through all the registered populate() and invokes them
  - Done automatically by the framework in the common code; Invisible to platform developer
- Step 3: Access API for a property:
  - Ex: FCONF_GET_PROPERTY(hw_config, uart_serial_config, base_addr)

fconf interfaces explained:

- FCONF_REGISTER_POPULATOR(a, b, c): A pre-processor macro
  - a : A name(string) which represents configuration type: Ex: “HW_CONFIG”
  - b: Used for debug purposes; Essentially a name representing the property namespace
  - c: populate() callback: In simple terms, a function which parses the dtb to retrieve the value of a property belonging to a node
  - “config_type” is associated with each dtb
  - Helps fconf to make sure populate() is invoked only on appropriate config dtbs
  - Ex: FCONF_REGISTER_POPULATOR(HW_CONFIG, uart_config, fconf_populate_uart_config);

```
#define FCONF_REGISTER_POPULATOR(config, name, callback)
        __attribute__((used, section(".fconf_populator")))
        const struct fconf_populator (name##__populator) = {
                .config_type = (#config),
                .info = (#name),
                .populate = (callback)
        };
```

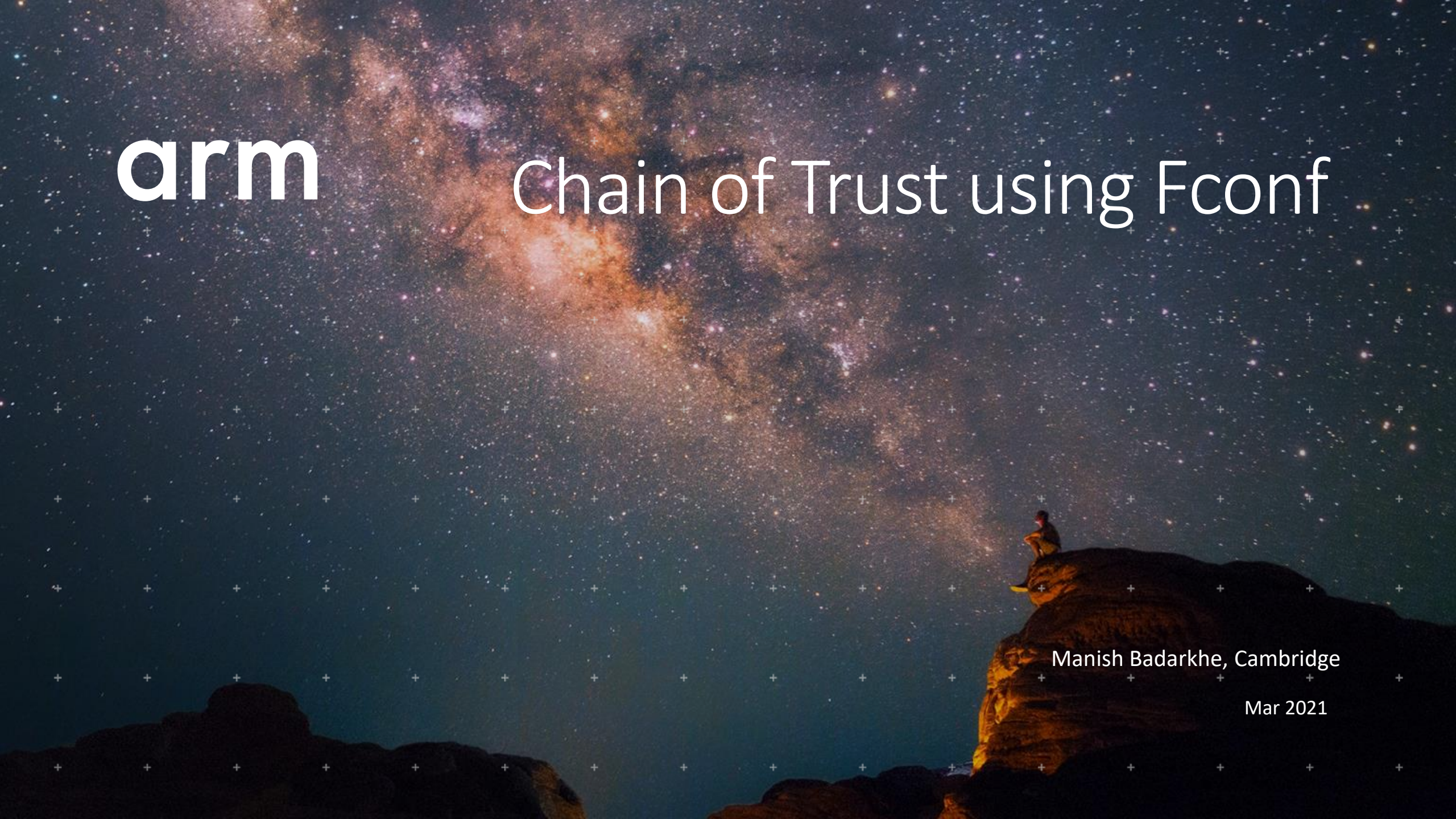
fconf interfaces explained:

- FCONF_GET_PROPERTY(namespace, sub-namespace, property): A preprocessor macro
  - Namespace loosely represents a collection of group of properties: Ex: “hw_config”
  - Sub-namespace loosely represents a group of related properties: Ex: “uart_serial_config”
  - Property is related to platform data/policy
  - Expands to **namespace__sub-namespace__getter(property)**
  - Responsibility of integrator to define this function call
  - Ex: base_addr = FCONF_GET_PROPERTY(hw_config, uart_serial_config, uart_base);

```
struct uart_serial_config_t {
        uint64_t uart_base;
        uint32_t uart_clk;
};

extern struct uart_serial_config_t uart_serial_config;

#define hw_config__uart_serial_config_getter(prop) \
                        uart_serial_config.prop
~
```

Leveraging fconf to move to dynamic configurations

- we aim to enhance trusted firmware(TF-A) by leveraging fconf framework.
- One such idea is to make various statically configured pieces of TF-A into dynamically configured components
- Example: IO Policies, Chain of Trust descriptors, hardware configuration of various devices such as UART, GIC etc.
- Don’t relying on hard coded compile time structures or macros
- Extract the platform specific configs using fconf and initialize various components in a BL image during runtime.



Chain of Trust using Fconf

Manish Badarkhe, Cambridge

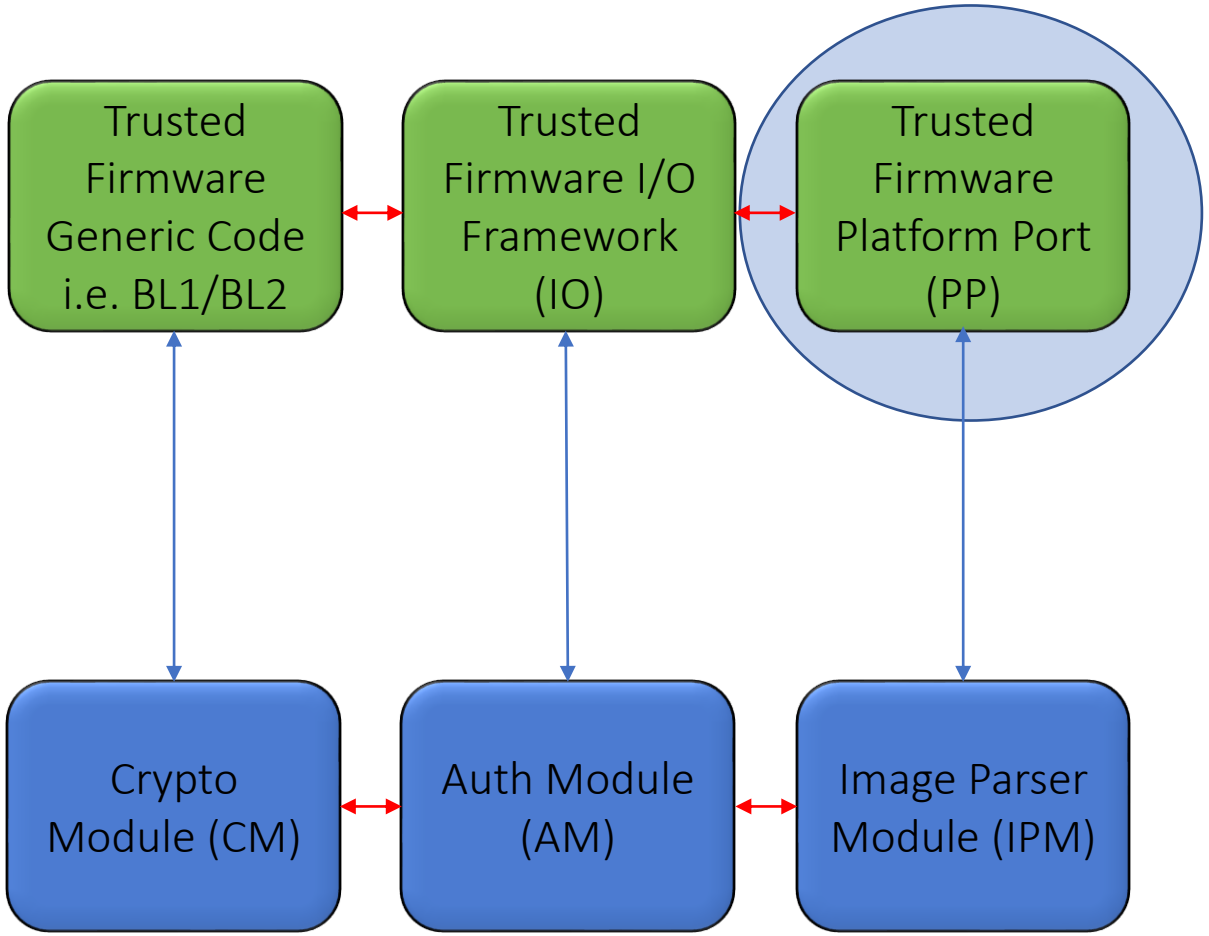
Mar 2021

Agenda

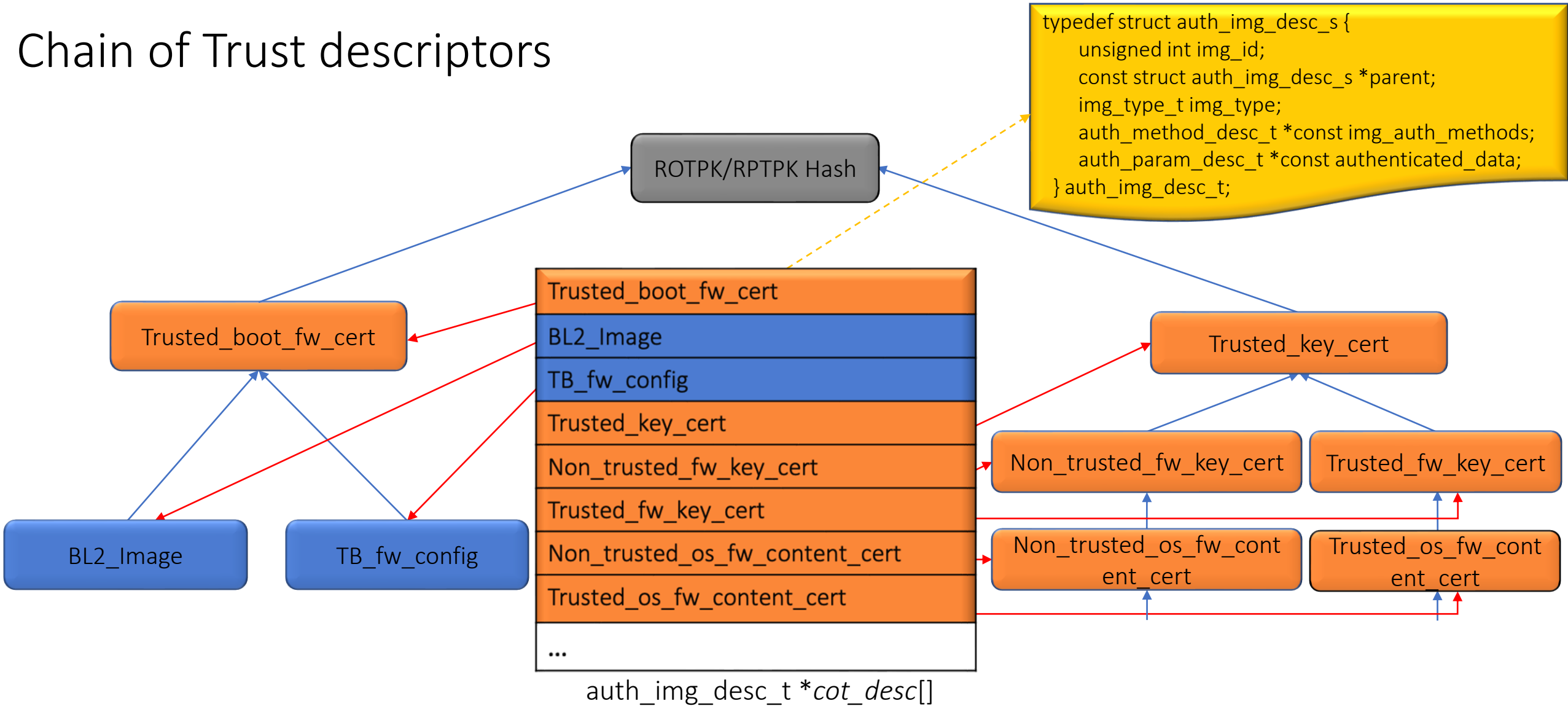
- Authentication Framework
- Chain of Trust statically implemented
- Chain of Trust in device tree
- Populate Chain of Trust
- Fconf APIs used in image verification
- Conclusion
- Resources

Authentication Framework

- Generic Code and I/O
  - load images using I/O framework
  - Authenticate images using AM
- Platform port
  - Specify CoT information for each image
  - Provision to get ROTPK or hash of it
- Modules
  - Verify CoT by using APIs exported by CM, AM and IPM



Chain of Trust descriptors



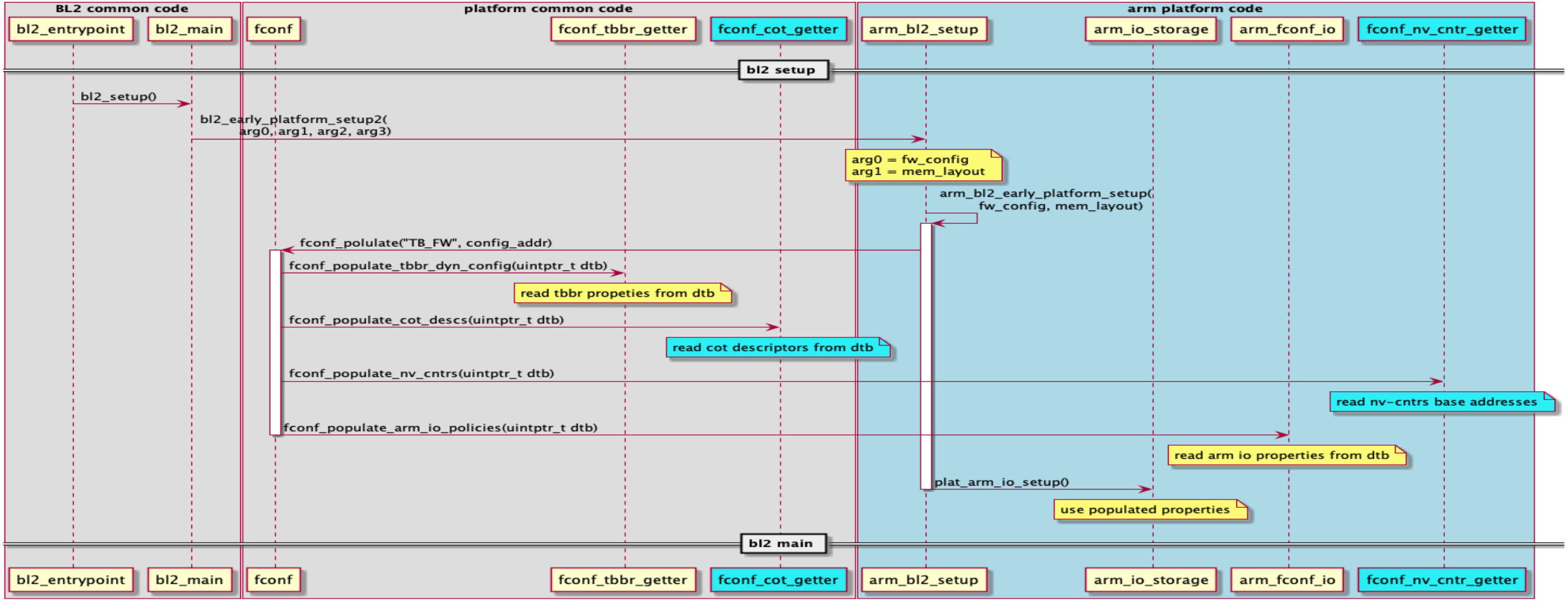
CoT device tree binding

```
manifests {
        compatible = "arm, cert-descs";

        trusted_boot_fw_cert: trusted_boot_fw_cert {
            root-certificate;
            image-id =<TRUSTED_BOOT_FW_CERT_ID>;
            antirollback-counter = <&trusted_nv_counter>;
            tb_fw_hash: tb_fw_hash {
                oid = TRUSTED_BOOT_FW_HASH_OID;
            };
            tb_fw_config_hash: tb_fw_config_hash {
                oid = TRUSTED_BOOT_FW_CONFIG_HASH_OID;
            };
            hw_config_hash: hw_config_hash {
                oid = HW_CONFIG_HASH_OID;
            };
            fw_config_hash: fw_config_hash {
                oid = FW_CONFIG_HASH_OID;
            };
        };

        soc_fw_key_cert: soc_fw_key_cert {
            image-id = <SOC_FW_KEY_CERT_ID>;
            parent = <&trusted_key_cert>;
            signing-key = <&trusted_world_pk>;
            antirollback-counter = <&trusted_nv_counter>;
            soc_fw_content_pk: soc_fw_content_pk {
                oid = SOC_FW_CONTENT_CERT_PK_OID;
            };
        };
        .....
}
```

```
images {
        compatible = "arm, img-descs";

        hw_config {
            image-id = <HW_CONFIG_ID>;
            parent = <&trusted_boot_fw_cert>;
            hash = <&hw_config_hash>;
        };

        tb_fw_config {
            image-id = <TB_FW_CONFIG_ID>;
            parent = <&trusted_boot_fw_cert>;
            hash = <&tb_fw_config_hash>;
        };
        .....
}
```

```
non_volatile_counters {
    compatible = "arm, non-volatile-counter";

    #address-cells = <1>;
    #size-cells = <0>;

    trusted_nv_counter: trusted_nv_counter {
        id  = <TRUSTED_NV_CTR_ID>;
        oid = TRUSTED_FW_NVCOUNTER_OID;
    };
    .....
}
```

Populate CoT from device tree



FCONF_REGISTER_POPULATOR(TB_FW, cot, fconf_populate_cot_descs)
FCONF_REGISTER_POPULATOR(TB_FW, nv_cntrs, fconf_populate_nv_cntrs)

Fconf APIs usage in image verification

- Get image descriptor
  - FCONF_GET_PROPERTY(tbbr, cot, img_id)
- Get base address of stored trusted NV-counter
  - FCONF_GET_PROPERTY(cot, nv_cntr_addr, TRUSTED_NV_CTR_ID)
- Get base address of stored non-trusted NV-counter
  - FCONF_GET_PROPERTY(cot, nv_cntr_addr, NON_TRUSTED_NV_CTR_ID)

Resources

- Binding Document

https://trustedfirmware-a.readthedocs.io/en/latest/components/cot-binding.html?highlight=chain%20of%20trust

- CoT in device tree implementation available in release v2.4

https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/tag/?h=v2.4

Conclusion

- Pros
  - Add image descriptors seamlessly with proper CoT.
  - Helps to add descriptors in more readable format.
  - Avoids using static initialized buffer and descriptors in the code.

- Cons
  - Need to allocate memory statically for MAX_NUMBER of descriptors in any case.