Device Tree lifecycle discussion

May 2020 - Joakim Bech, François Ozog, Ilias Apalodimas

Linaro

History

Early Device Tree environments

- Originally created by Open Firmware.
- Few uses cases involving firmware (actually firmware may have been non existing at all).
- Linux owned all resources and all hardware.
- DT configurations mainly (only?) used by Linux kernel.
- DT was a pure hardware configuration.

Recent years environments

- DTB has been used by some vendors as kernel driver parameters DB which proved to be volatile and introduced instability in Linux kernel (booting a new kernel with an old kernel DTB).
  We reached some maturity 2, 3 years ago and we are at a point where pushing the DTB out of kernel entirely can actually facilitate enforcing stability.
- (Android) Device Tree Overlays (DTO).
- Firmware installed secure world services (PSCI, OP-TEE etc) need to advertise themselves in the DTB.
- Need to pass other information between firmware, not only HW configuration (think the “chosen” node).
- Linux kernel isn’t owner of all hardware and resources any longer.
- Some devices and peripherals needs to be shared with firmware.

Who owns hardware ?

Firmware changing role: Trusted substrate is living next to the main OS

- **Role was**: prepare the system to load the OS.
- **Role becomes**: active trust services for applications or to limit hardware usage.
  - Not UEFI runtime: Secure state with private memory and may be devices.
  - S-EL2 a game changer for trust applications and services
    - Automotive ISO24089
    - Open Late Binding (in TrustZone @Arm, in Management Engine microcontroller@Intel)
    - “Trustlets” like micropayments and DRM
    - Regulatory policies enforcement (video-surveillance)
    - Dedicated secure storage for insurance company non-repudiable logs

Device assignment for applications (DPDK) or VMs

- No issues for enum capable buses (PCI) but a problem for platform devices.
- Need a way for sysadmins (not fw/kernel developer) to assign a device.
  - A single device is made of nodes, pinctrl, irqs, clocks, regulator with phandles ...
  - No metadata in DTB to identify what makes a “device” (actually, there is even no metadata to identify handle “pointers”.

DT Lifecycle work items

Objectives

- Define clear authority boundary between firmware and OS.
- Increase stability and security for OS (DT is not a driver parameter DB).
- Define collaboration between firmware components.
- Simplify DT to deal with HW diversity.
- Simplify DT for device assignment (DPDK or Hypervisor).

Design goals

- Applicable to any firmware.
  - U-Boot SPL, TF-A, Coreboot ...
  - OP-TEE, Trusty ...
  - U-Boot (EFI or non EFI), AVB? LinuxBoot?
- Applicable to any OS.
  - Linux, BSD, Android.
- Leverage/enhance existing technologies.
  - Android Device Tree Overlays (DTO).

Deliverables

- Specifications: DT, EBBR, AVB?, SMC calls?
- Upstream patches: U-Boot, OP-TEE, TF-A/M, Linux, Zephyr.

Simplified boot chain in our current devices

An even more simplified dummy example to make it easier to discuss, here “A” could be secure side firmware, “B” could be non-secure boot loader and “O” could be the OS.

Linaro

Different types of problems

1. DTS-file “duplication”.
2. Missing firmware handover and runtime rules.
3. Trusting DTB.

Problem 1 - DTS duplication

- The same type of files are located in several different gits.
- In a boot chain A → B → O, developers for all of them needs to synchronize their work.

Fictional example where things are not in sync:

foodev-in-A.dts

```
[...]
        uart1@: serial@0x02020000 {
                compatible = "org, foodev-2000-uart";
                reg = <0x02020000 0x4000>;
                status = "okay";
        }

        reserved-memory {
                #address-cells = <2>;
                #size-cells = <2>;
                ranges;

                secure-side-device@0x99990000 {
                        reg = <0x00000000 0x99990000 0 0x01000000>;
                        no-map;
                };
        };
[...]
```

foodev-in-O.dts

```
[...]
        uart1@: serial@0x02020000 {
                compatible = "org, foodev-2000-uart";
                reg = <0x02020000 0x4000>;
                status = "disabled";
        }

        reserved-memory {
                #address-cells = <2>;
                #size-cells = <2>;
                ranges;

                secure-side-device@0x00001111 {
                        reg = <0x00000000 0x00001111 0 0x01000000>;
                        no-map;
                };
        };
[...]
```

(Yes, addresses are not aligned ... this is an oversimplified example just for the discussion)

Problem 1 - DTS duplication

The example on previous slide can lead to situations like:

- Developer for “A” wonder why UART is disabled when the system is up and running, he for sure enabled it in his DTS file.

- Developer for “O” get sudden crashes/memory corruptions? Why? “A” is a runtime firmware and have mapped another region in his setup.
    - Developer “A” changed the reserved memory at some point, but didn’t change the reserve-memory in “O”.

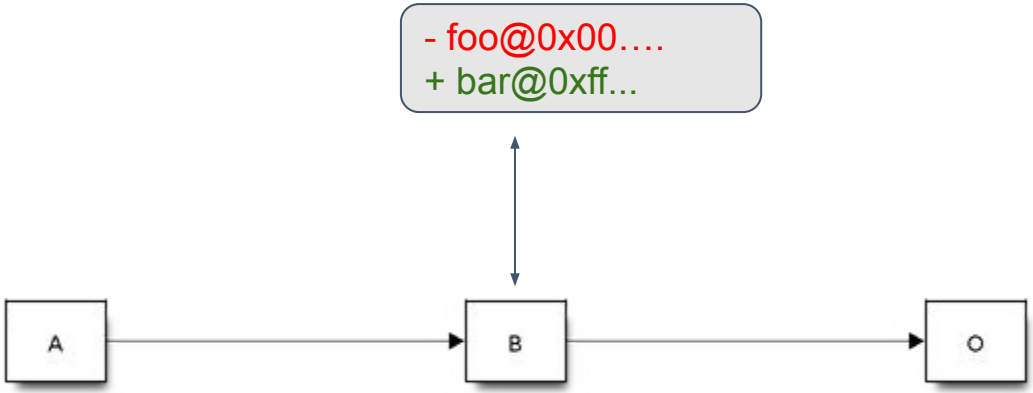
Why is it a problem?

- It’s a very fragile setup, things can easily break.
- You as a the one writing the DTS file never really have the true picture of how it’ll look like when the system is up and running.
- Changes needs to be synchronized and preferably accepted/merged at the same time.
- Duplication of DTBs is not always complete across projects. Many boards cannot boot a Linux OS using the U-Boot provided DTB

Problem 2 - handover and runtime rules

When passing a DTB from one firmware to another there are no hand-over rules.

- In a boot chain A → B → O, anyone can **read, write, add, remove** nodes, configurations etc.
- “O” might have written a DTS, so it expects that node "foo" is there, but "B" actually deleted it in runtime and added a node "bar" instead.
- This also makes it hard to understand whether a DTB has been **modified intentionally or by an attacker**?



Problem 3 - Trusting DTB

- Systems cannot be trusted if the the firmware hasn't been signed and verified, which is the reason we implement secure boot / chain of trust.

- The Device Tree signature support story is not coherent
  - U-Boot have "Verified Boot", that can verify any type of binary
  - AOSP/Android has implemented AVB.

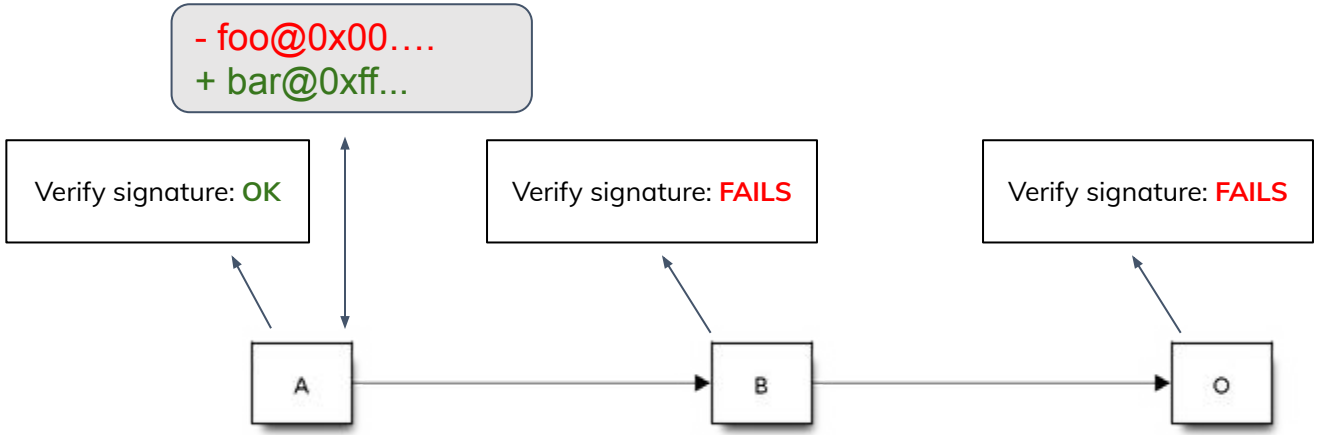
- Running a system with unverified DTB files is like opening up pandora's box to hackers:
  - Change addresses ...
  - Add nodes ...
  - Remove nodes ...

- Side Channel Attacks
  - For this discussion, please keep in mind that even a Chain of Trust verified firmware **is** susceptible to a run-time SCA attack (glitching for example).

Problem 3 - Signed DTB

All DTB's needs to be signed and verified, which leads to ...

- You **cannot** do runtime modifications (add, remove, modify) to DTB's since then the signature verification would fail at the next boot stage.
- Implicitly that would mean that the "chosen" node cannot be used as today. It'll also affect how you're dealing with Device Tree Overlay (*).



(*) With implicit trust it could work, but in the most secure case, then it will not work.

Problem 3 - Signed DTB

How can you design signature verification of DTBs? A few examples ...

Single DTB

- A single **immutable** DTB is provided to the first stage boot loader (on secure side)
- Each and every boot stage verify the DTB.

Multi DTB (plagued with “problem 1” and “problem 2” on previous slides)

- Many different **immutable** DTBs are provided/used in each stage of the boot.
- Key management becomes a challenge, you need to multiple priv/pub keys.
- If you need runtime modification, then you basically need to sign the DTB again before handing it over. Private key management becomes a nightmare.

Bake a DTB into a firmware-blob (which is verified)

- Still needs to be **immutable**, i.e., runtime modifications after verification cannot happen (the DTB is in this case is implicitly verified).

Problem 3 - Signed DTB

Conclusions

- You **should** sign and verify DTB's just as any other firmware, otherwise you're running an untrusted system!

- The DTB has to be **immutable**.
  - This affects discussions we've had around passing (TPM) "measurements" between firmware using Device Tree.
  - Things like the "chosen" node, cannot be used as of today (*).
  - DTO must be considered.

- Splitting up the DTB into "same same but different" DTBs make signature/verification story complicated.
  - I.e., "Problem 1" in this deck is actually also an issue when it comes to running signed DTB's.

- Run-time attacks when the system if fully up and running is hard to mitigate.

(*) With implicit trust it could work, but in the most secure case, then it will not work.

Call to action

Questions we should ask ourselves

- Do we actually need to do runtime modifications to the DTB blob?
- Do we need to look at DTBs from another angle? One part covering hardware configs and one covering software configs?
  - Do we need to sign everything?
- Are the other ways to achieve the same? Measured boot?
- Why can't we have a single DTB given to the first stage boot loader and then this is propagated all the way through the boot chain?
  - Memory constraints?
  - Hot-plugging?
  - Device Tree Overlay came by because a single representation was problematic.
- Wouldn't a generic "DT git" help quite a lot to overcome the challenges presented in this deck?
- How would we have designed something like Device Tree if we had never done Device Tree and had no knowledge about it? Starting point would probably be the first boot loader and not Linux kernel.

Thank you

Backup slides

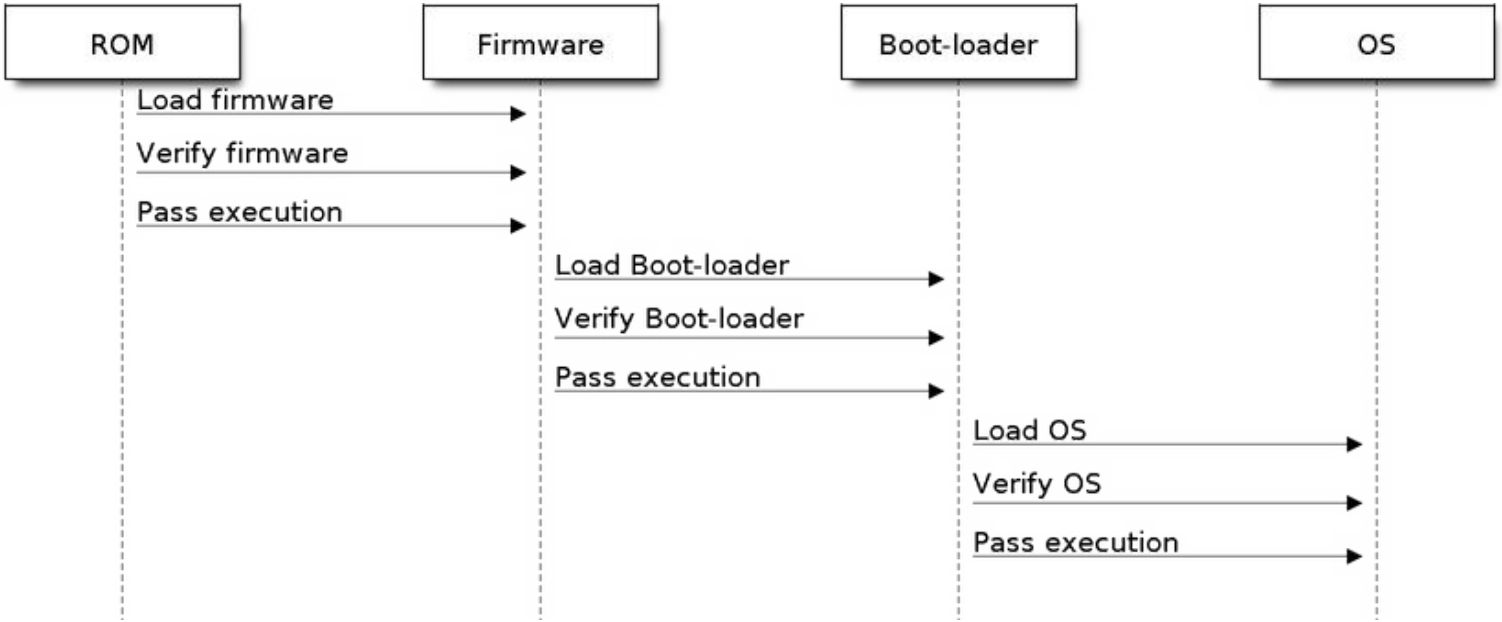
DTB lifecycle constraints

- Reference DTB for the platform need to be updatable.
  - Probably on a file system (SSD, NvME...).
  - Can embed it in FIP but very impractical to change for operational situations.
- DTB cannot be loaded too early 256KB of SRAM, 30KB DTB ...
- No common DT repo, need to start for kernel.org produced DTBs.

- Signature
  - No standard for DTB signature.

- Versioning
- Probably need to include kernel version (actually the version translated into an “uint”) in the DTB and the produced fragment so that everyone can take appropriate action.
  - Using kernel version sounds bad but I think that’s the project that have the biggest legacy.
  - Over time, central repo version (uint = ORIGIN + function(central repo version)) should replace kernel’s one. ORIGIN is the kernel version number we chose to do the switch.

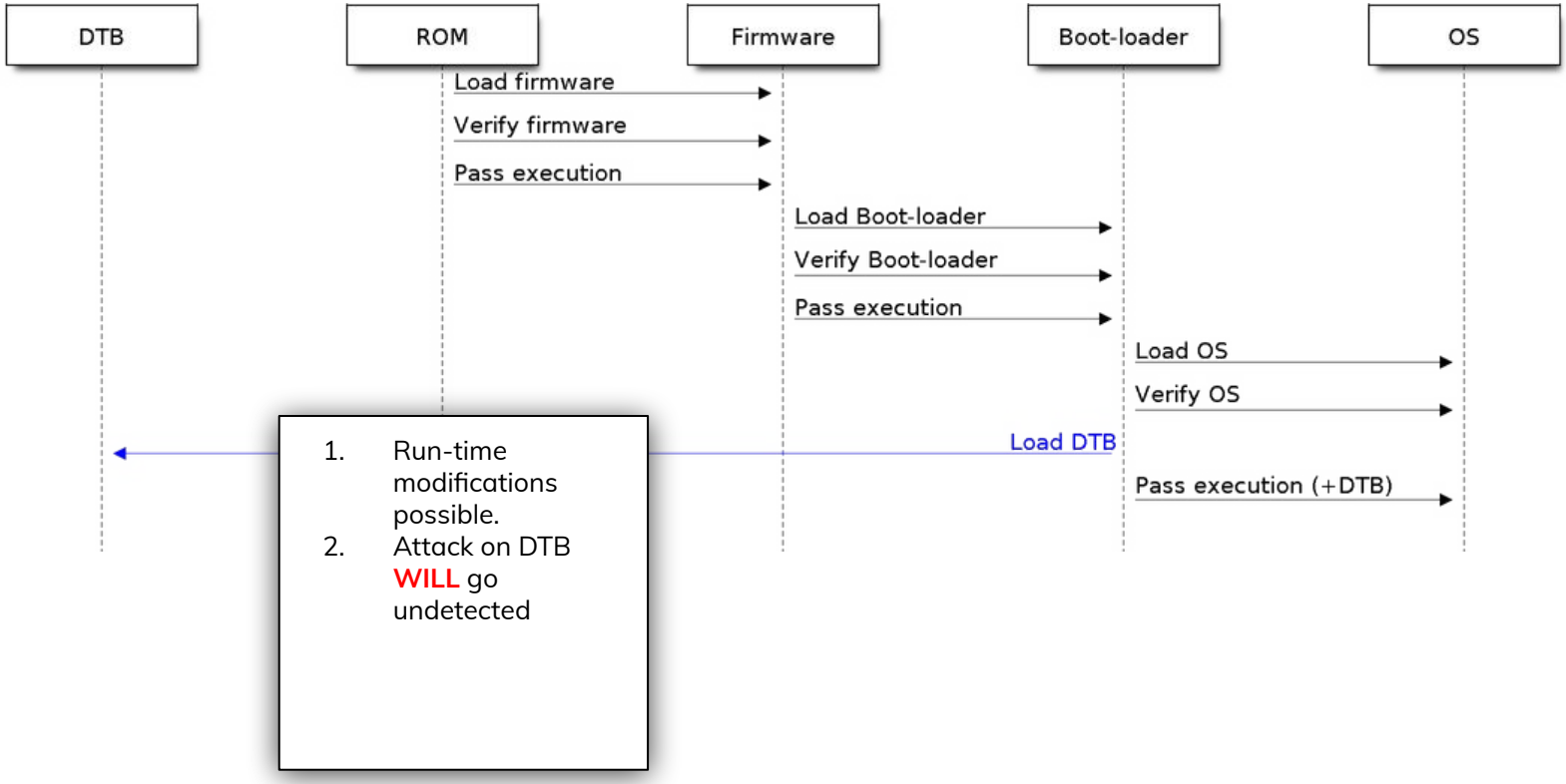
Backup#1 - Traditional chain of Boot



Backup#1 - Traditional chain of Boot

- DTB isn’t considered, quite unlikely scenario on many embedded devices today

Backup#2 - CoT without DTB verification



Linaro

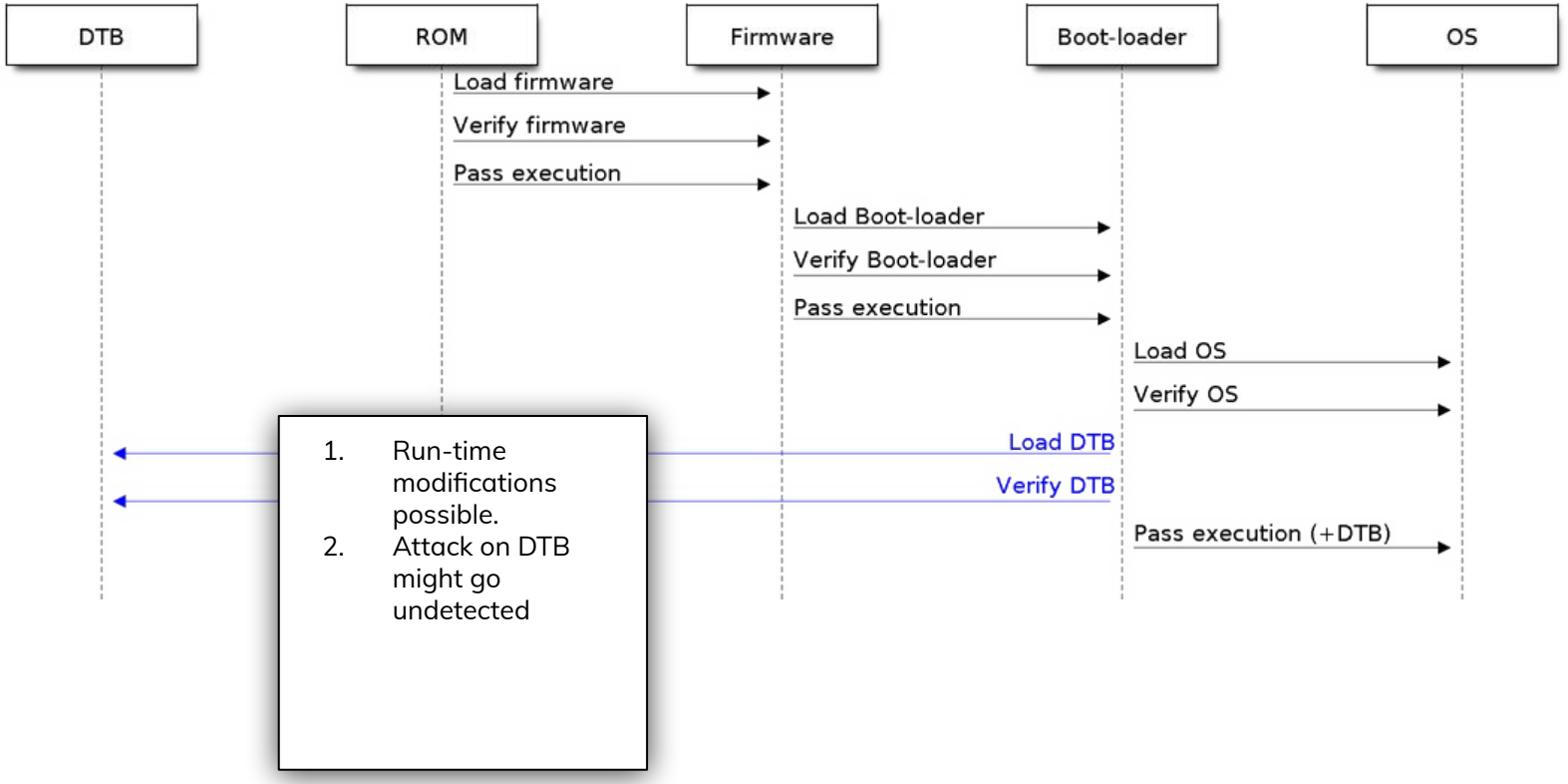
Backup#2 - CoT without DTB verification

- DTB is loaded late in the boot
- This is typical setup when no sort of DTB signature verification is enabled
- DTB run-time modifications are possible

Attacks?

- Tamper the DTB binary
- Run-time modifications (during boot and when the system is fully up and running)

Backup#3 - CoT with late DTB verification



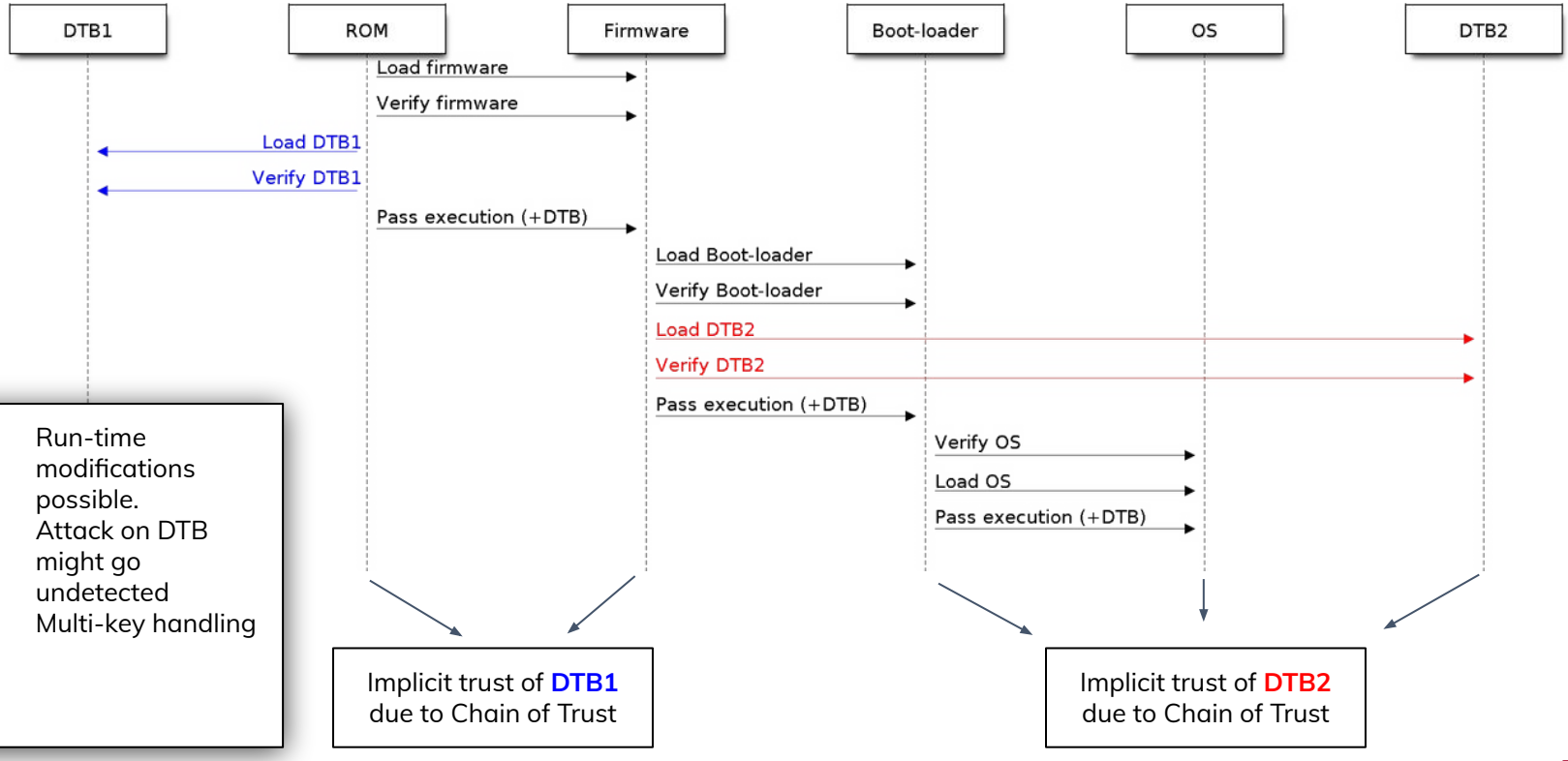
Backup#3 - CoT with late DTB verification

- Basically the same as a typical setup, but with some sort of DTB signature verification is enabled.
- Late verification

Attacks?

- Run-time modifications (during boot and when the system is fully up and running)

Backup#4 - CoT with multiple DTBs



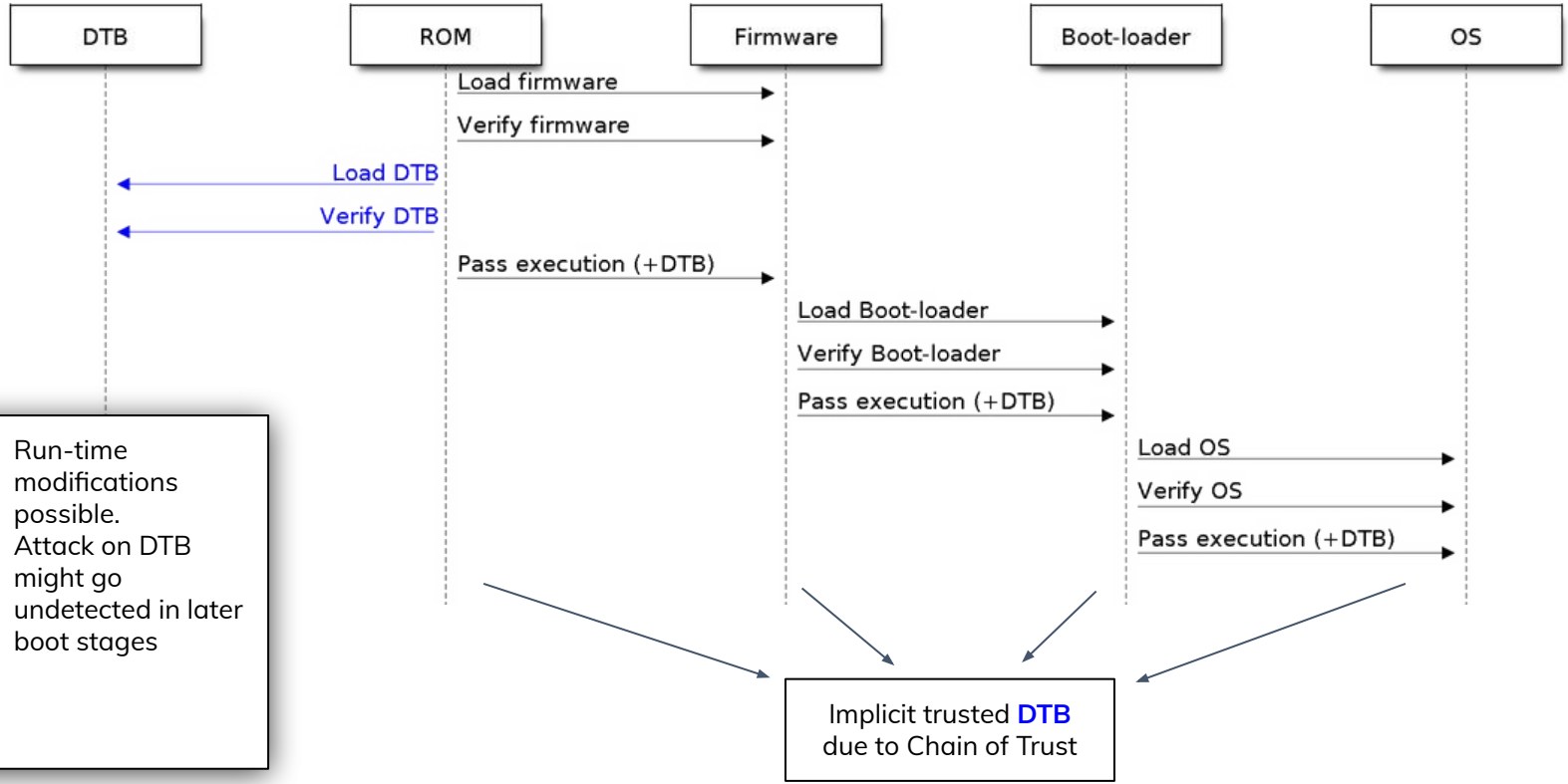
Backup#4 - CoT with multiple DTBs

- Also somewhat common if we set a side verification
  - Different boot stages loads “their own” DTB, might or might not pass it on
- Dual verification means that different stages will need to have different public keys (if they weren’t signed with the same private key of course)
- DTB run-time modifications are possible (to some extent)

Attacks?

- Run-time modifications (during boot and when the system is fully up and running)

Backup#5 - Trad. CoT with DTB loaded once



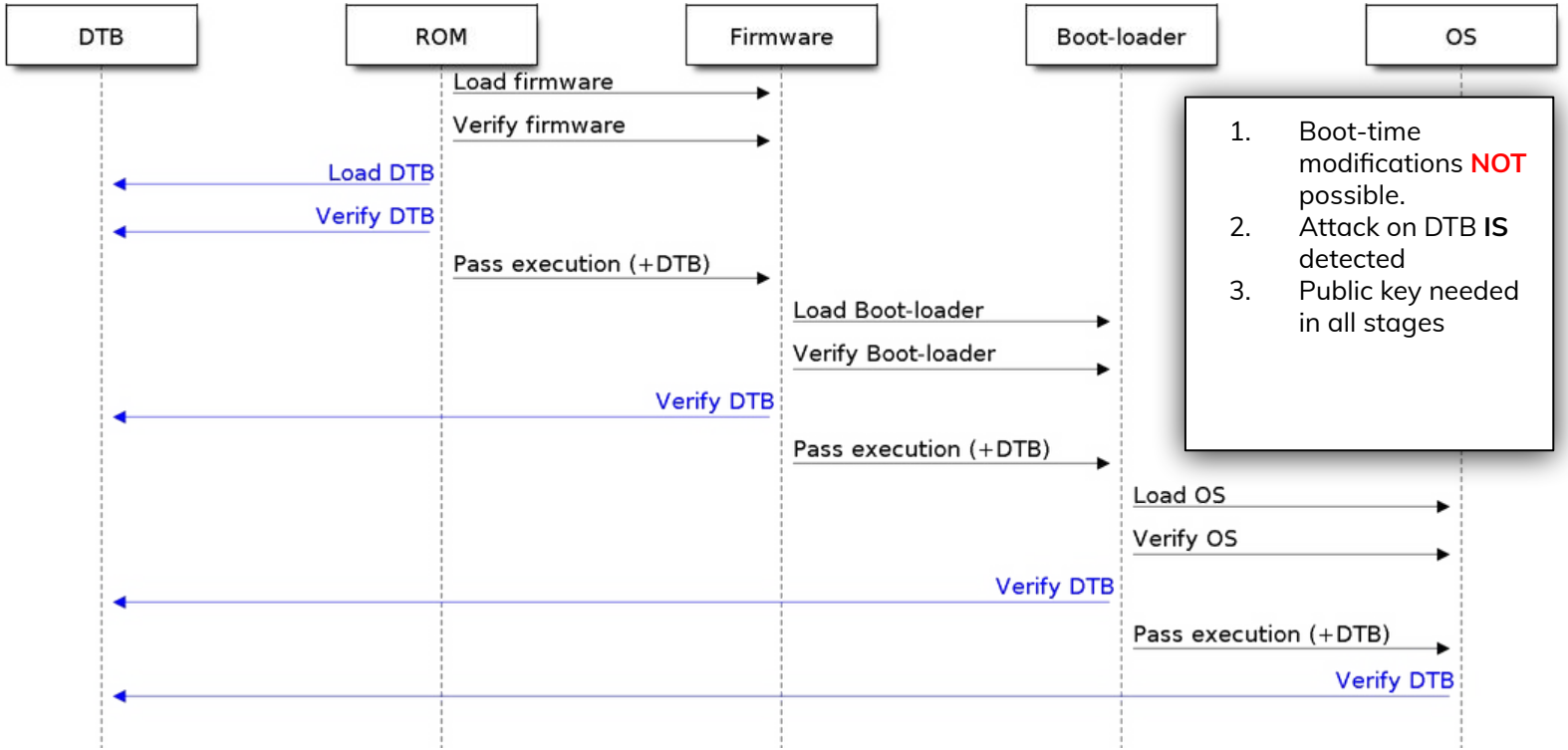
Backup#5 - Trad. CoT with DTB loaded once

- First thing running on the device loads and verifies the DTB
- Implicit trust of DTB based on normal Chain of Trust
- DTB run-time modifications are possible

Attacks?

- Run-time modifications (during boot and when the system is fully up and running)
  - Exploit in run-time firmware can tamper with DTB being passed on
  - Example: Boot exploit in OP-TEE (TEE core), the payload modify the DTB in memory
  - Again, remember Side Channel Attacks!

Backup#6 - CoT with individual DT verification



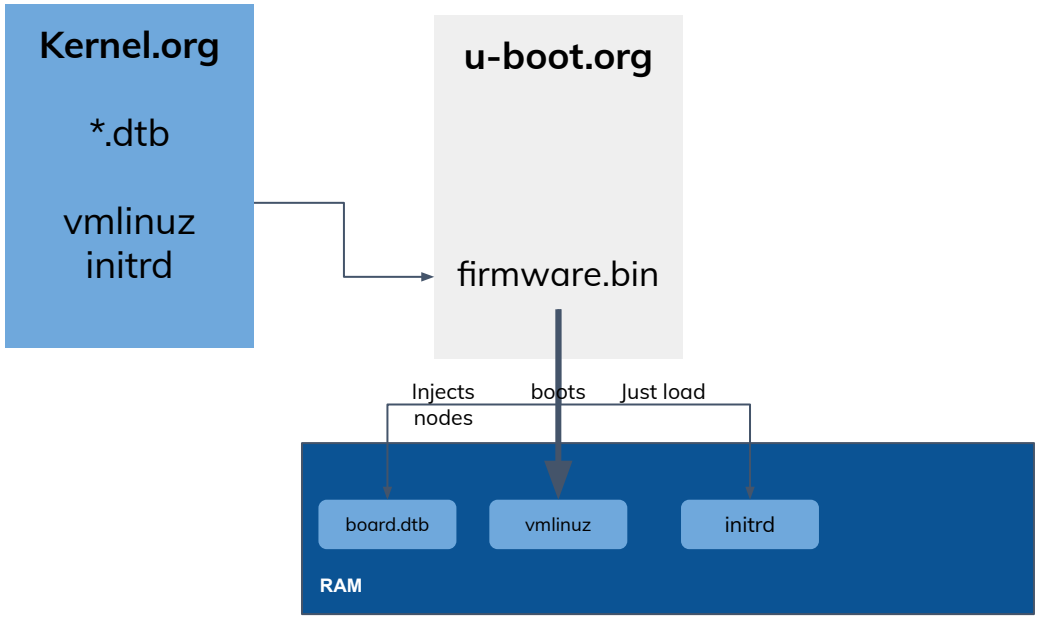
Backup#6 - CoT with individual DT verification

- DTB verifications doesn’t rely only on Chain of Trust or a “one time DTB verification”
- Each stage in the boot verifies the DTB on their own.
- Each stage needs access to the public key in one or another way
- Impossible to modify DTB’s (even intentionally)

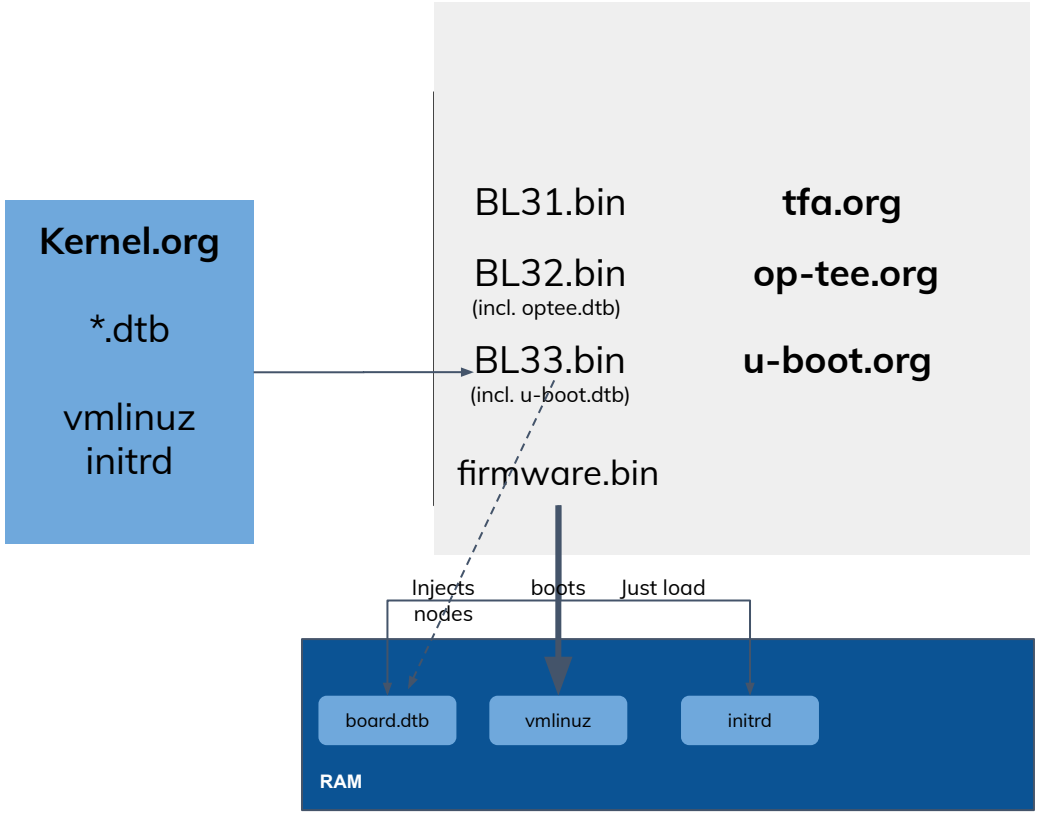
Attacks?

- Run-time modifications (when the system fully up and running)

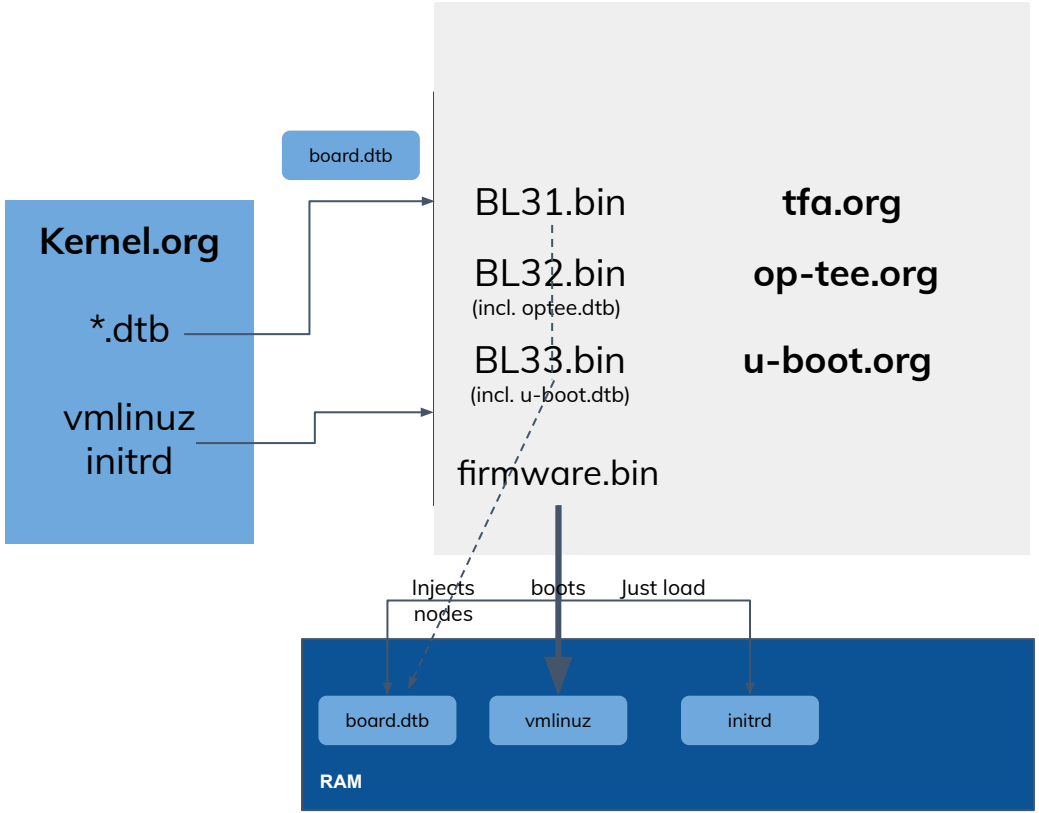
Current situation, Linux kernel vision



Current situation, firmware vision



DT Lifecycle



Kernel *.dtb constraints:

- No “firmware” nodes (PSCI)
- No chosen

Board.dtb (signed by kernel)

- Loaded by BL33
- Issue new SPCI call to signal intention to boot a payload with board.dtb
- TF-A, OP-TEE, OP-TEE “reply” by sending signed DTB fragments (PSCI+RAM) (OP-TEE service) that may also enable or disable nodes
- Linux receives the board.dtb along with all signed fragments/overlays and decide what to do.
- Mechanism valid regardless of code base (TF-A + OP-TEE + OP-TEE, U-Boot, SPL + OP-TEE + U-Boot; TFA + Trusty + ...)

Linaro