THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

Embedded Linux Conference

Secure Boot for Embedded Linux

Explained in Simple Words

Roy Jamil
Embedded Systems Trainer, Ac6

About me – Roy Jamil

- Embedded Systems Trainer at **Ac6**
- PhD in real-time embedded systems
- I teach Linux, RTOS and security
- Provide training and support on Zephyr and Linux

About Ac6

- Specialized in embedded systems training for over 20 years
- Focus areas include RTOS, Linux, Security, FPGA and more
- IDE development : SW4STM32, Workbench for Zephyr
- Training: Secured Embedded Linux Platform Build
  - Secure boot
  - OP-TEE
  - LSM (SELinux...)

Why this talk exists

Secure Boot is not actually hard.

It looks hard because the way it is usually explained involves:

- Heavy cryptography vocabulary
- Vendor-specific flows and marketing names
- Diagrams with 30 boxes and 50 arrows

Goal: Explain Secure Boot in simple terms

Agenda

- Introduction
- Signature
- Chain of trust
- Secure World
- Conclusion

The threat: what could go wrong?

Imagine your embedded device boots Linux every morning.

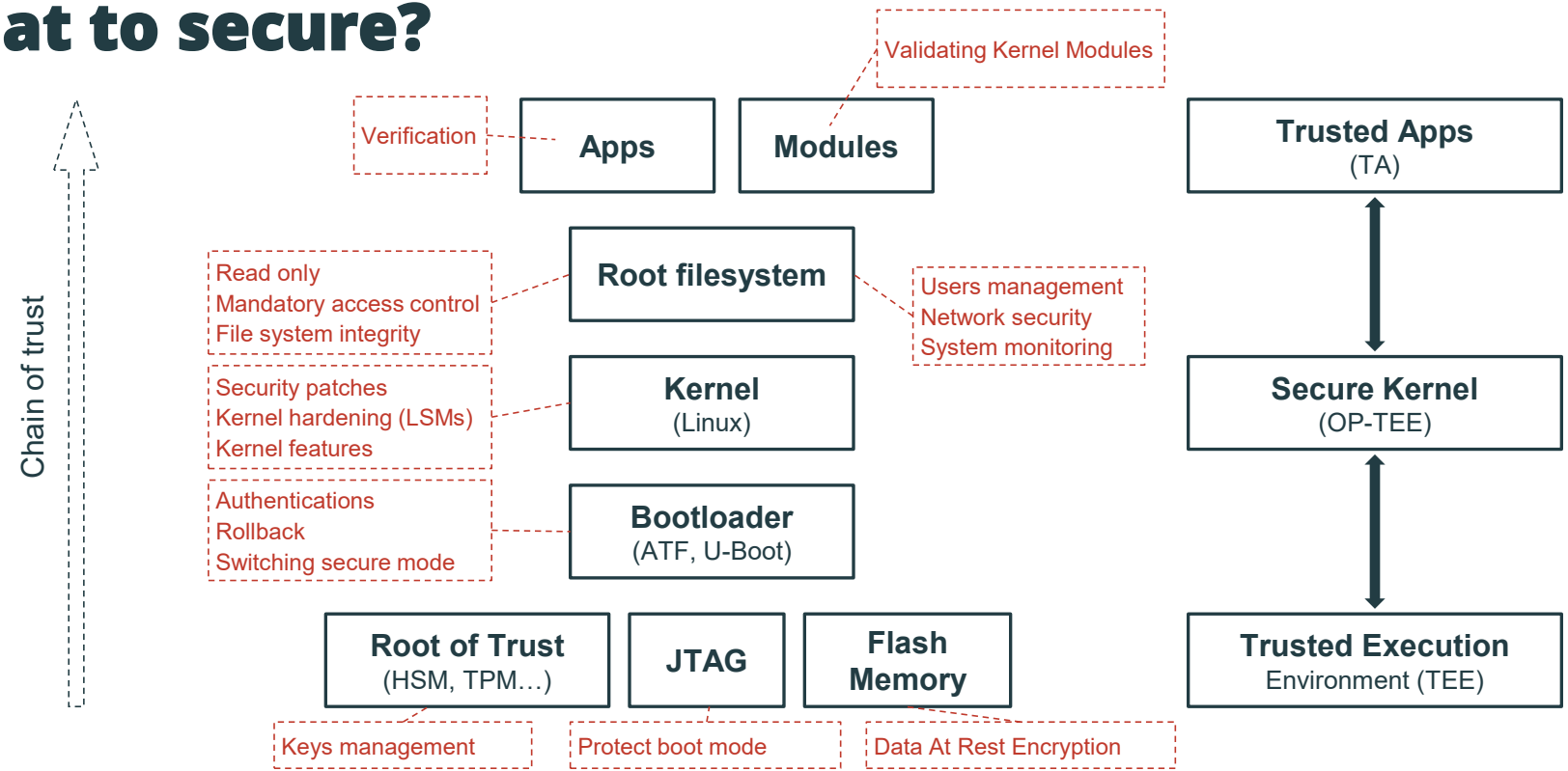
Nothing checks the software before it runs.

An attacker who can reach the storage chip can:

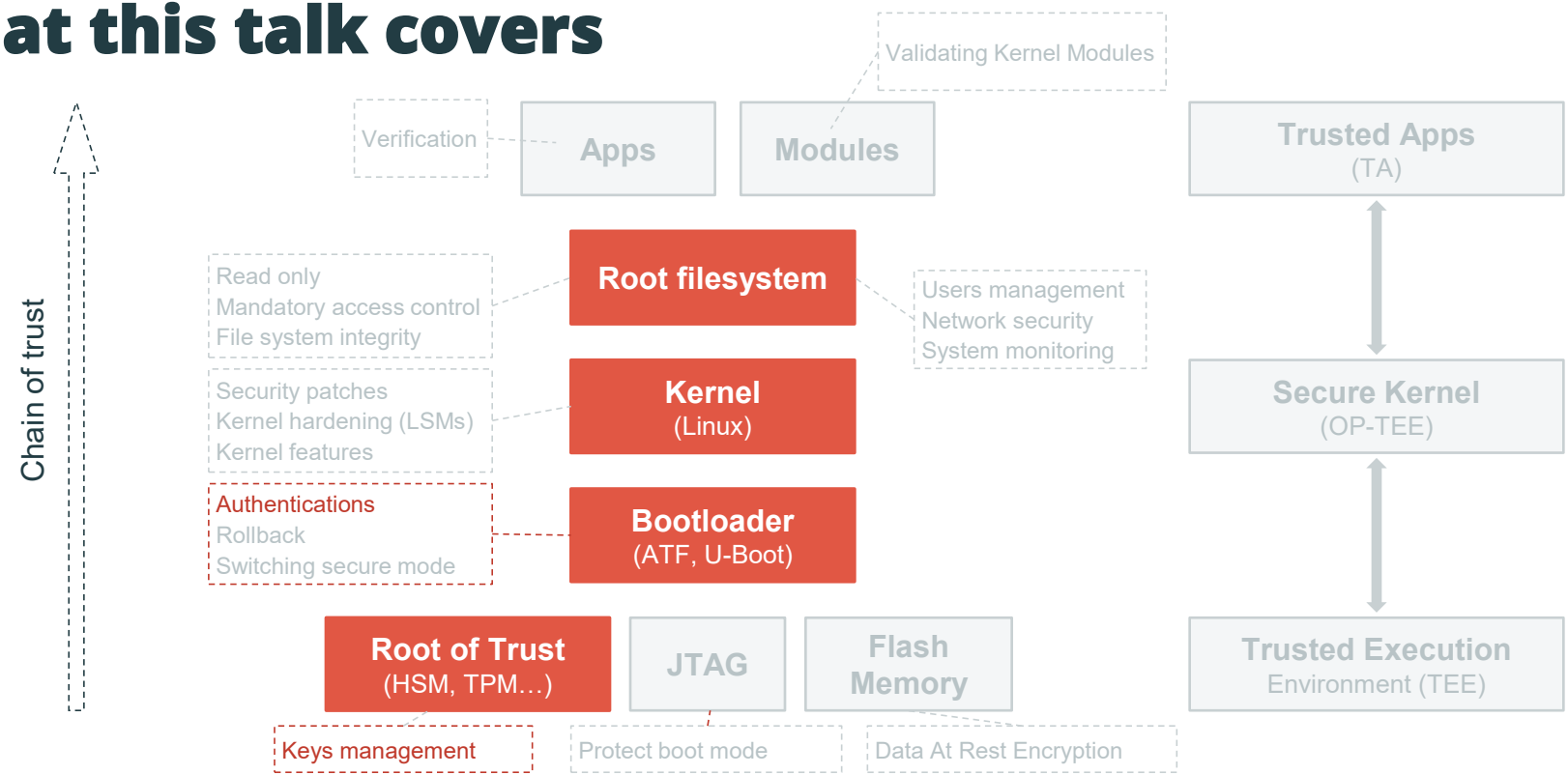
- Swap your kernel for theirs
- Patch the bootloader to skip authentication
- Install a persistent backdoor that survives reboots

And the device boots happily, as if nothing happened.

What to secure?



What this talk covers



What Secure Boot really is

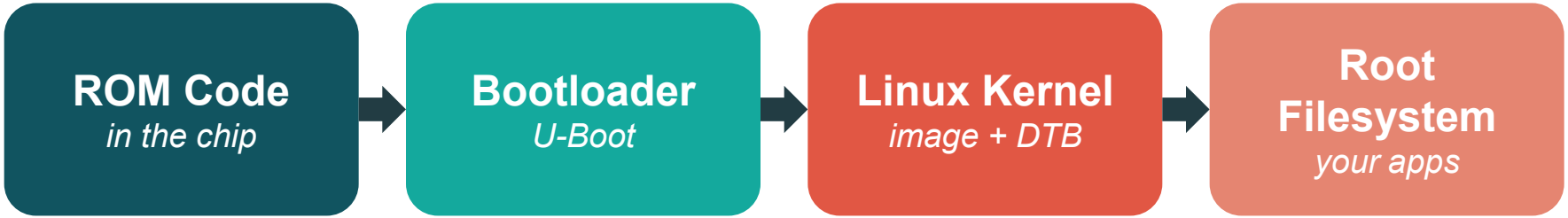
"Only run software you approved."

That is it. Everything else is plumbing.

- From the very first instruction the CPU executes
- All the way up to Linux userspace
- Every piece of code is checked before it runs

Not the same as: disk encryption (confidentiality)

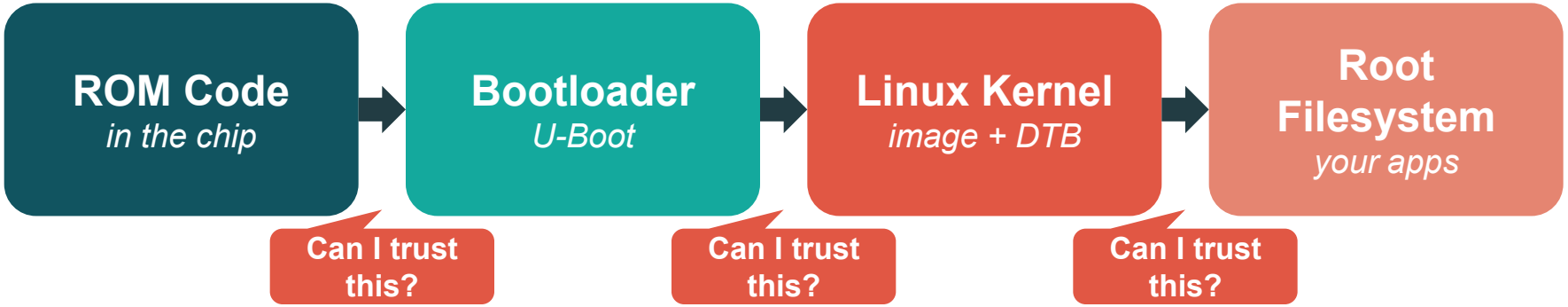
The boot chain, WITHOUT trusted boot



Each stage loads the next from storage and jumps to it.

No one is asking "is this software trustworthy?"

The boot chain, WITH trusted boot



Signature

Three building blocks

1
Hash

A fingerprint of a file
Same file = same fingerprint
Tiny change = totally different

2
Keys

Private key (you keep secret)
Public key (you share)
They are mathematical twins

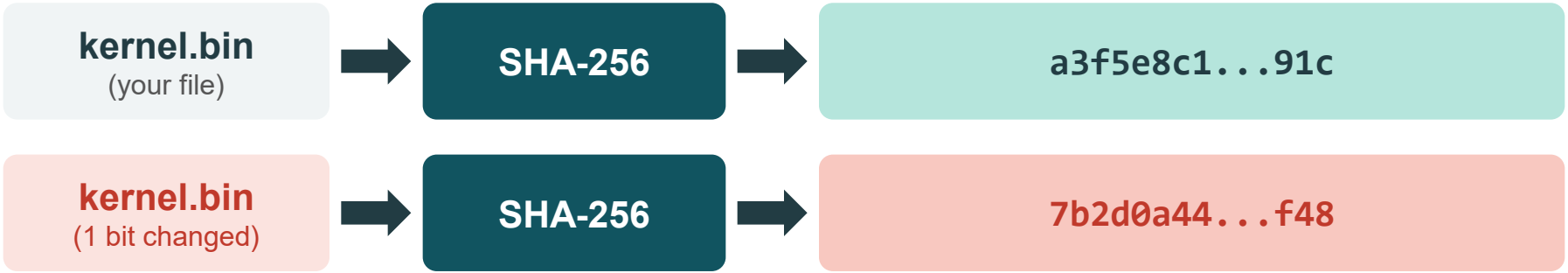
3
Signature

Proof that someone with the private key vouches for this file.

That is it. We will use these three again and again.

Hash = digital fingerprint

- Feed any file into a hash function, get a fixed-size string
- Same file in, same hash out, every time
- Change one bit, the hash changes completely
- Cannot go backwards: given the hash, you cannot rebuild the file



Two keys, two jobs

One signs. The other verifies. **Easy to check, impossible to forge.**

Private key

Like a signing stamp only you own

Used to sign files
Lives in your vault or HSM
If it leaks, game over

Public key

Like the imprint everyone can recognise

Used to verify signatures
Burned into the device at the factory
Safe to share, that's the point

Lock with the public key, unlock with the private

Secure Boot does the **opposite**: sign with the private key, verify with the public key on the device

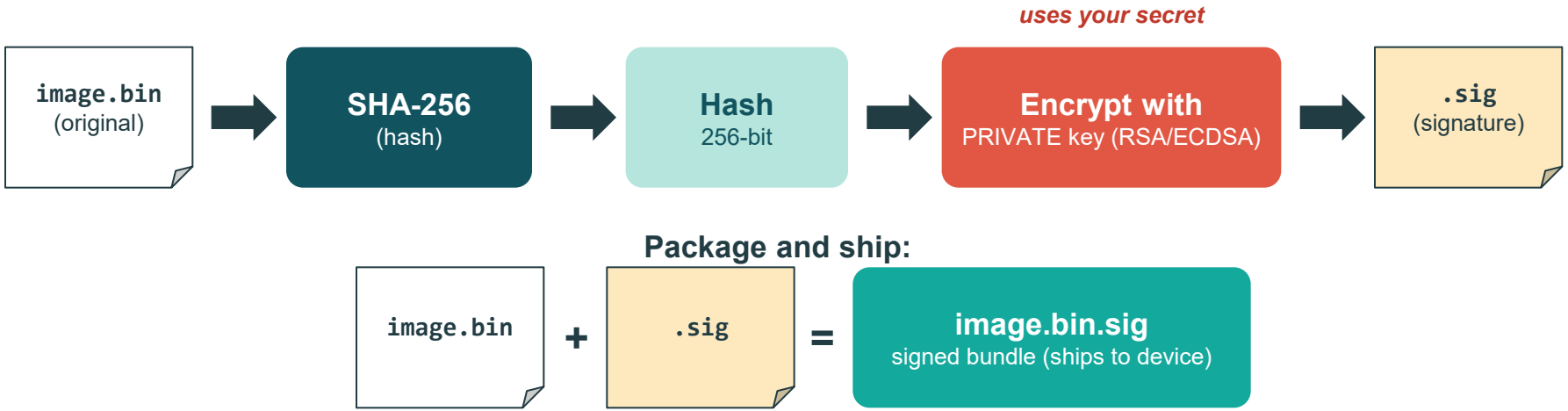
Signature = a sealed envelope

How to sign a file:

1. Compute the **hash** of the file
2. Encrypt the hash with your **private key**
3. That encrypted hash **is** the signature

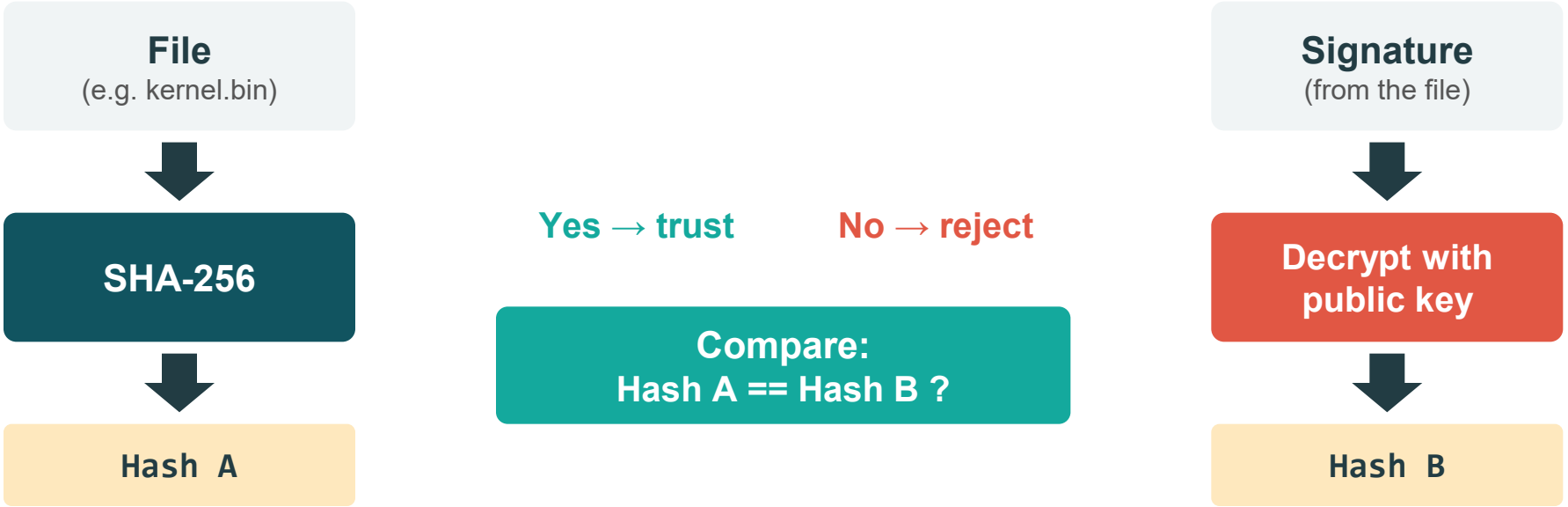
Think: a wax seal on an envelope. Only the seal's owner can make it.
Anyone with the matching pattern can recognize it as real.

Signing: how a signature is made



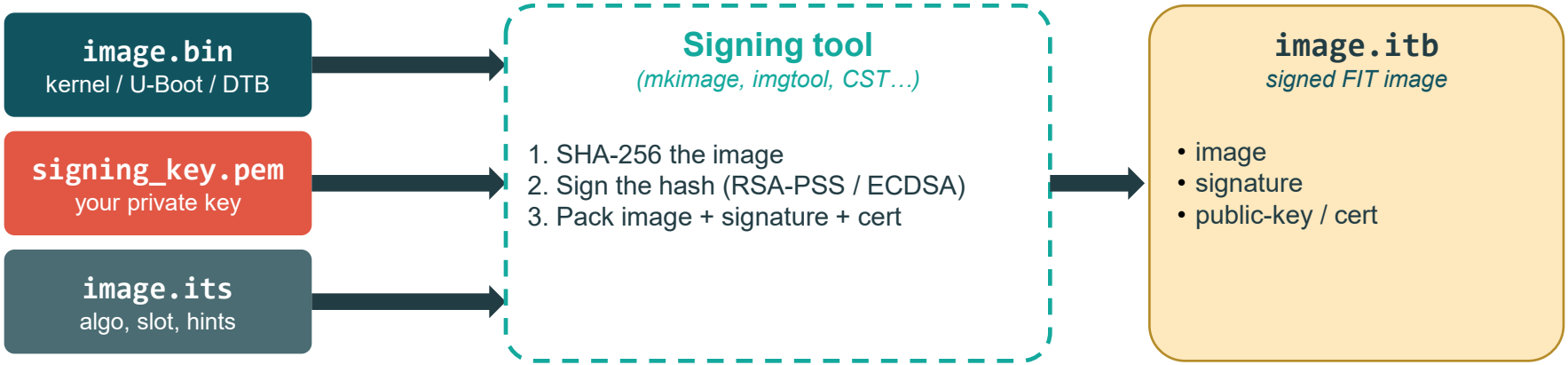
The private key never leaves your private build server. Only the signature and public key travel with the device.

Verifying a signature



What an embedded signing tool does

Signing, wrapped in a tool: `mkimage`, `imgtool`, NXP `CST`, ST `STM32_SigningTool`.



What's actually inside that .itb

A **FIT image** (Flattened Image Tree) is one binary file that bundles everything U-Boot needs to boot Linux, plus the signature that proves it.

image.itb *(binary FIT tree)*

```
images {
  kernel-1   { … }
  fdt-1      { … }
  ramdisk-1  { … }
}
```

```
configurations {
  default =
  "kernel+fdt";
```

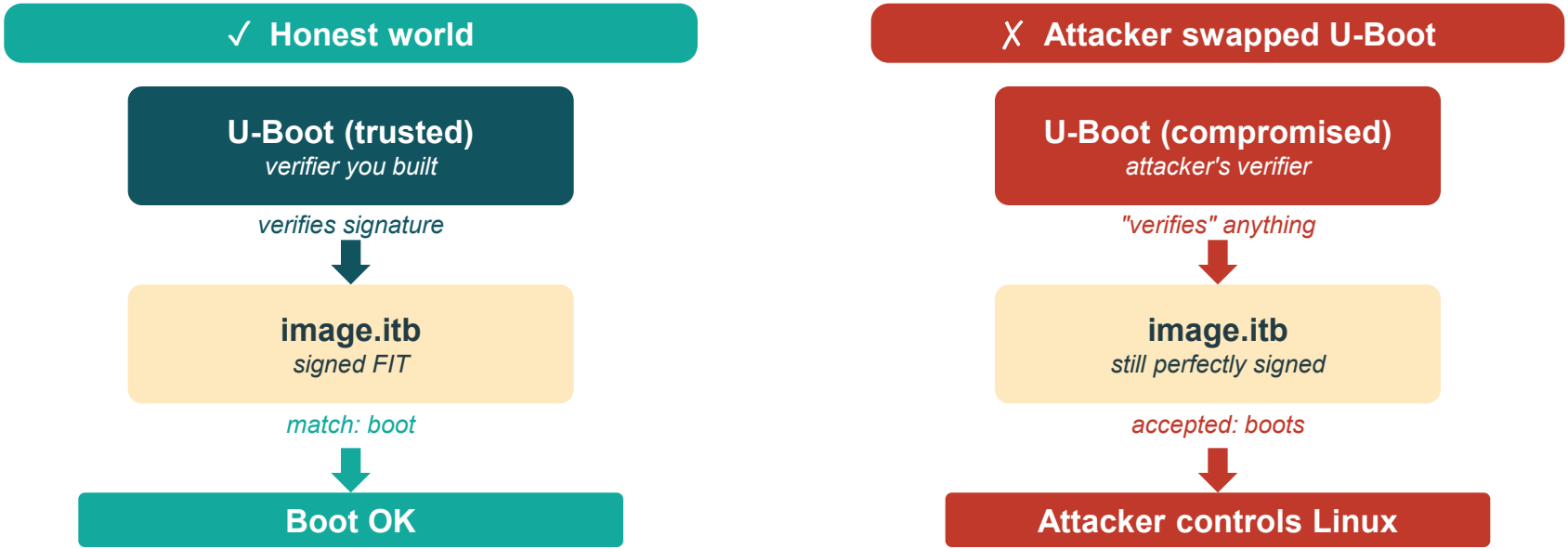
```
signature {
  algo = sha256,
         rsa2048;
  key-name-hint;
  value = …
}
```

Binary, not source
You write image.its, mkimage compiles it.

U-Boot reads it natively
bootm $loadaddr selects a configuration.

One signature covers it all
Kernel, DTB and ramdisk verified together.

A signed FIT alone is not enough



Chain of trust

The Root of Trust (RoT)

"Trust has to start somewhere."

Every chain has a first link. You cannot verify the verifier with itself.

The Root of Trust is:

- A piece of code or data that we **assume** is trustworthy
- Stored where an attacker cannot easily change it
- The starting point everything else gets verified against

Where the trust actually lives

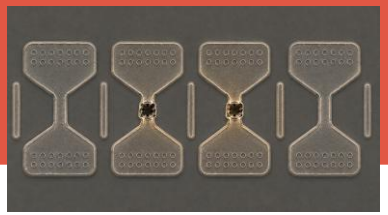
Two pieces of hardware do all the heavy lifting:

ROM (Boot ROM)

Code etched into the SoC at the factory
Cannot be changed after fabrication
Runs first, every boot
Contains the very first verification logic

OTP fuses

Tiny one-time-programmable bits
Burn once, read forever
Store the hash of your public key
ROM trusts whatever this hash says



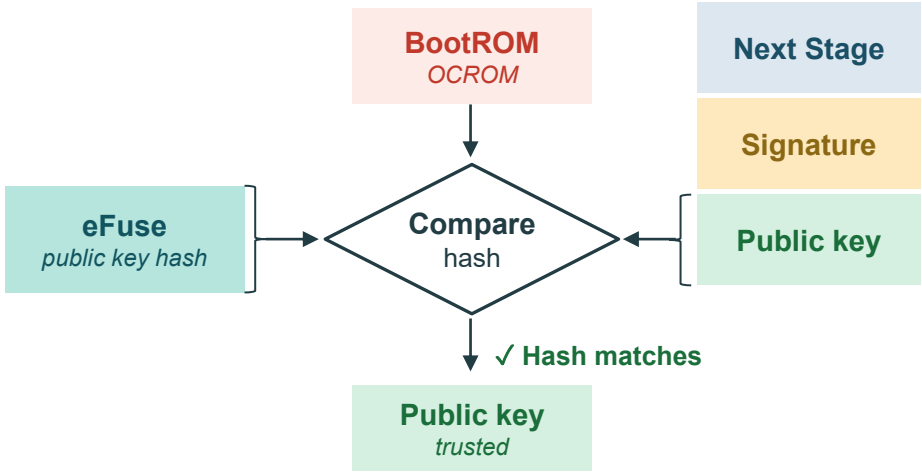
How does the ROM know which key to trust?

It doesn't store the public key.
It stores a **hash of the public key** in eFuses, and recomputes it at boot.

RSA-2048 public key
256 B
too big for eFuses

vs

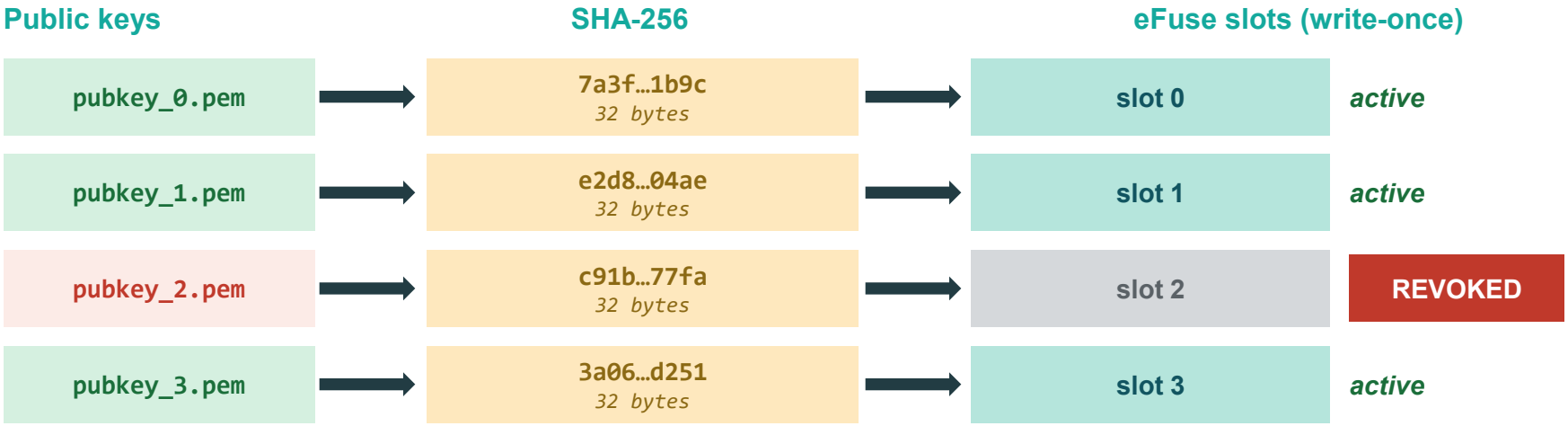
SHA-256 hash
32 B
fits in eFuses ✓



- Hash is one-way
  - So revealing it is safe
- Several slots
  - You can revoke a key, use another

Why four hashes, not one

Most SoCs reserve **several fuse slots**. If a key leaks, you burn a revoke bit on its slot



Boot accepts an image signed by ANY non-revoked slot.
Lose a key, revoke its slot, ship images signed by the next one.

Signing

Sign at build time. Burn the pubkey hash at factory. (Verification runs at every boot.)

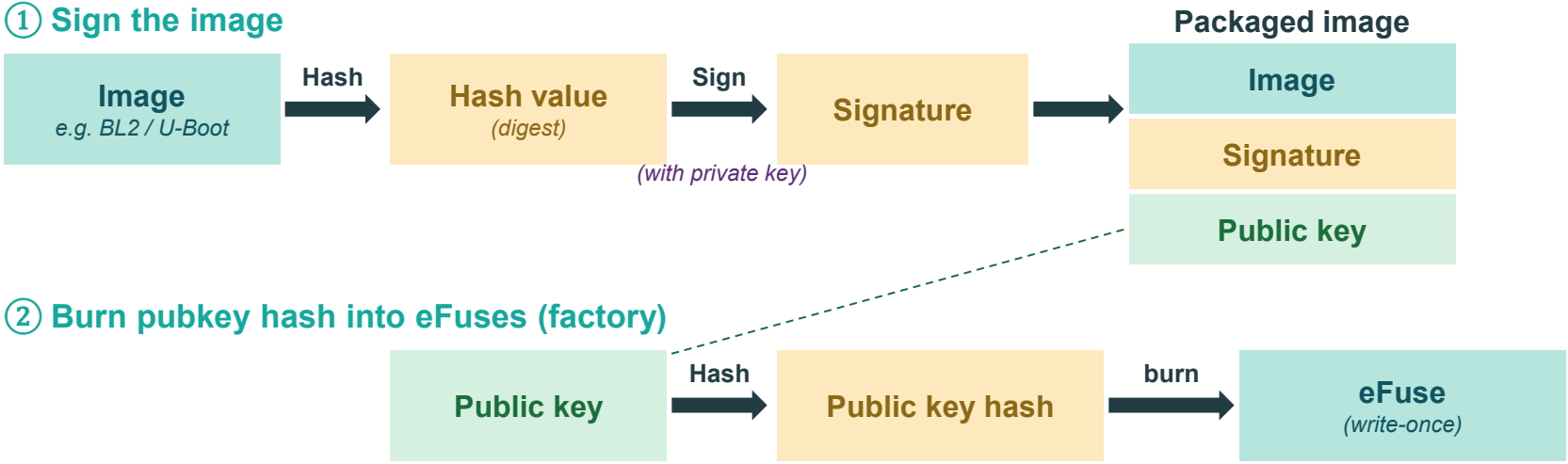
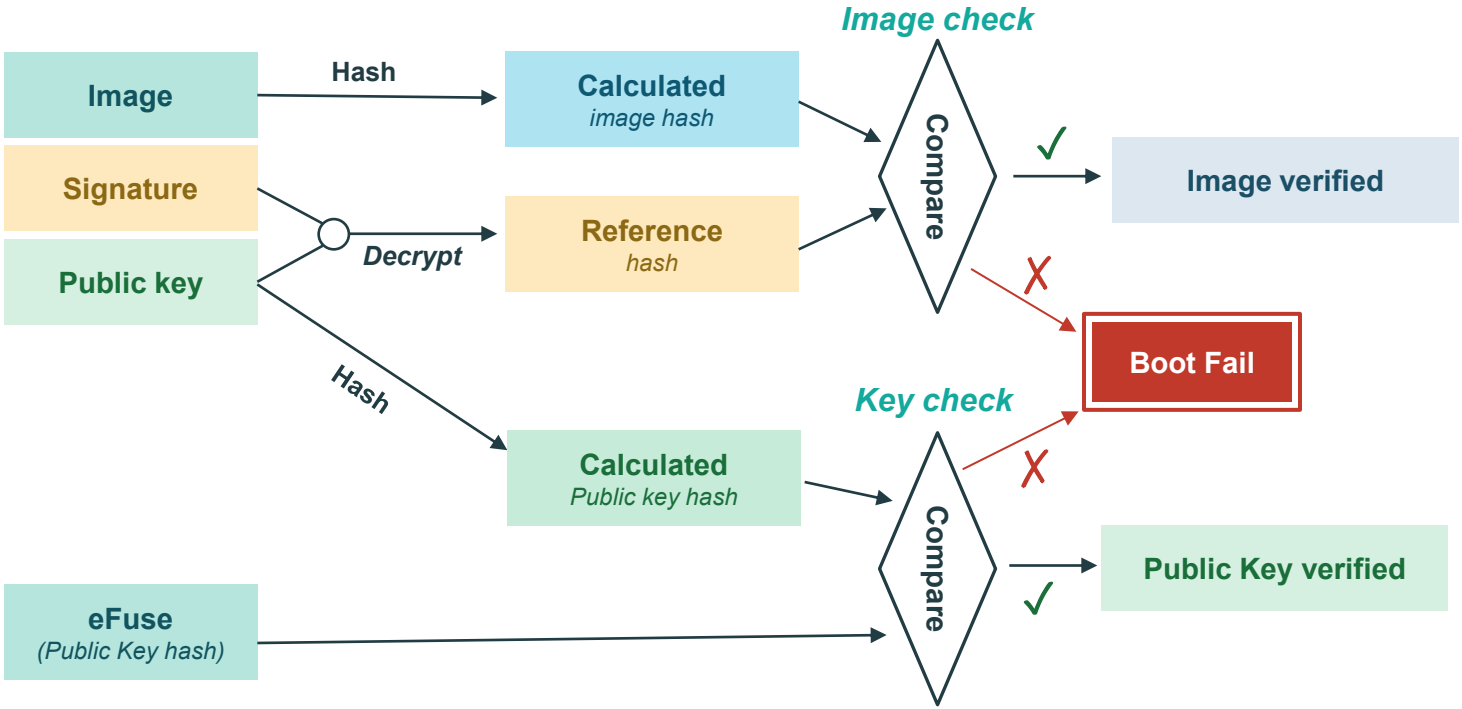


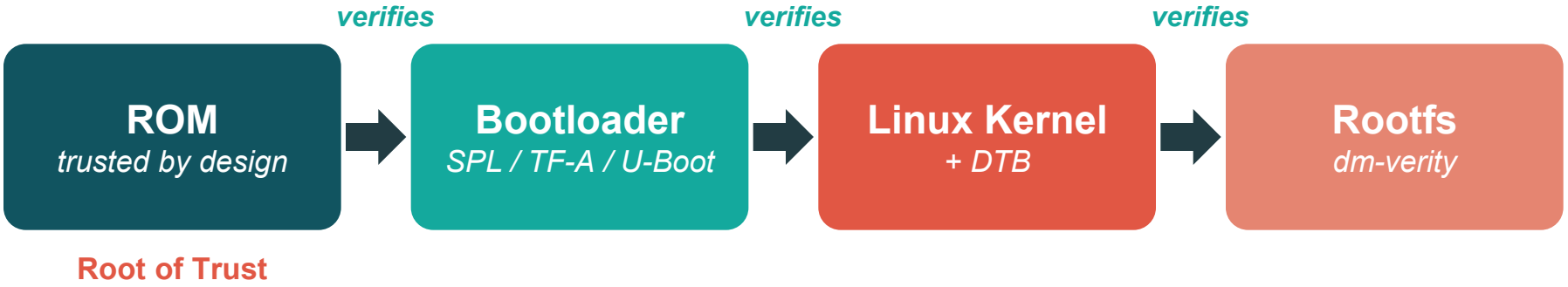
*Image ships with its signature and public key. The device only carries the **32-byte hash** in its fuses.*

Runtime verification: Image & Key



Chain of trust: each link verifies the next

Same boot chain as before, but now each stage verifies the next before jumping to it:



Trust starts in hardware and propagates one step at a time.
Break any link and the chain is broken.

Step 1 of 3: ROM verifies the first loader

The CPU powers on. ROM runs first. What it does:

1. Find the first-stage loader on flash (SPL, FSBL, BL1, depending on the SoC)
2. Read the signature attached to it
3. Hash the loader image, decrypt the signature with the public key from OTP
4. Hashes match? Jump to it. Mismatch? Halt or fall back.

This is the only step the silicon vendor controls. Everything after, you control.

Step 2 of 3: First loader verifies TF-A and U-Boot

On modern ARM platforms the bootloader is now split into two parts:

- **TF-A (Trusted Firmware-A)** runs in the Secure World, sets up the platform
- **U-Boot** runs in the Non-Secure World, loads the kernel

The first loader signs and verifies each of these images:

- Each image has its own signature in a Firmware Image Package (FIP)
- Public keys are embedded in the loader image (already verified by step 1)
- Any mismatch stops the boot immediately

Step 3 of 3: U-Boot verifies the kernel

U-Boot uses a packaging format called **FIT image**:

- One single file bundling kernel + device tree (DTB) + optional initramfs
- Each component is hashed; one signature covers them all

If you forget to sign the DTB, you forgot half your trust.

(The DTB tells Linux what hardware exists, where to mount root, kernel cmdline)

Beyond the kernel: dm-verity

Secure Boot stops at the kernel. **dm-verity** extends the chain to the rootfs
It only works with **read-only** filesystems like squashfs

- Root hash signed, passed to the kernel via cmdline
- Tampered block, kernel refuses the read
- dm-verity operates below the filesystem layer, on raw blocks.

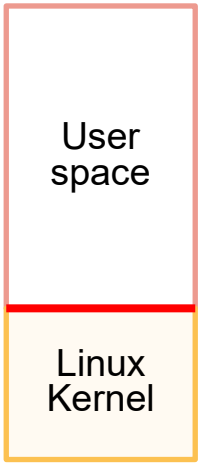
Read-only rootfs + dm-verity = trust reaches userspace.

Secure World

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

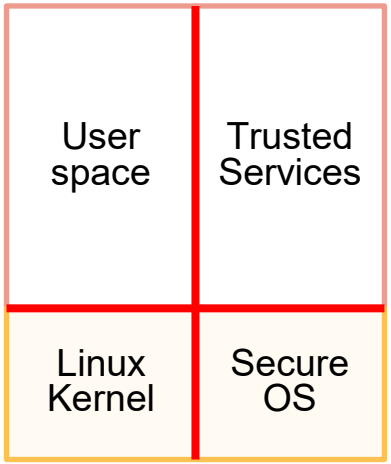
Embedded Linux Conference

Execution environment setups



Normal OS

Isolates userland from Kernel

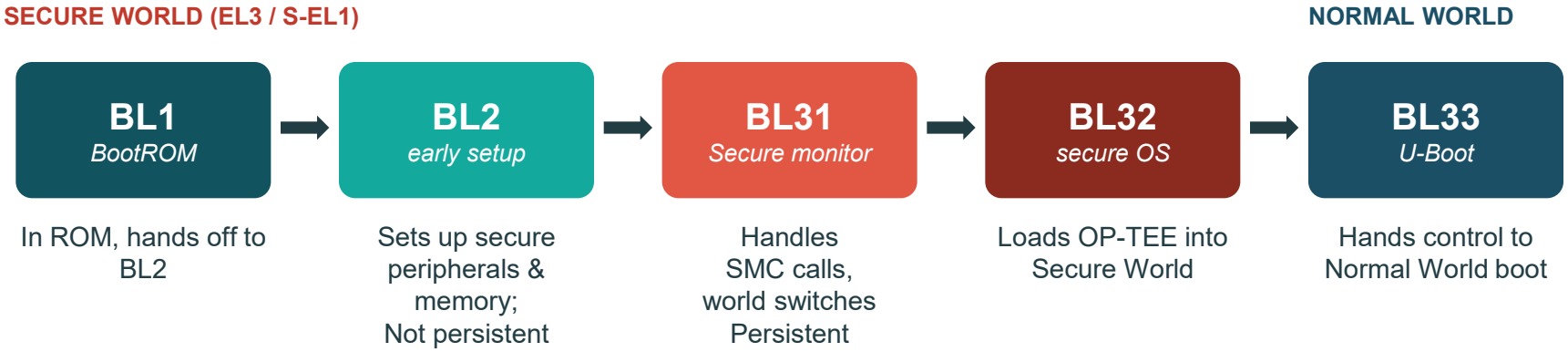


TrustZone

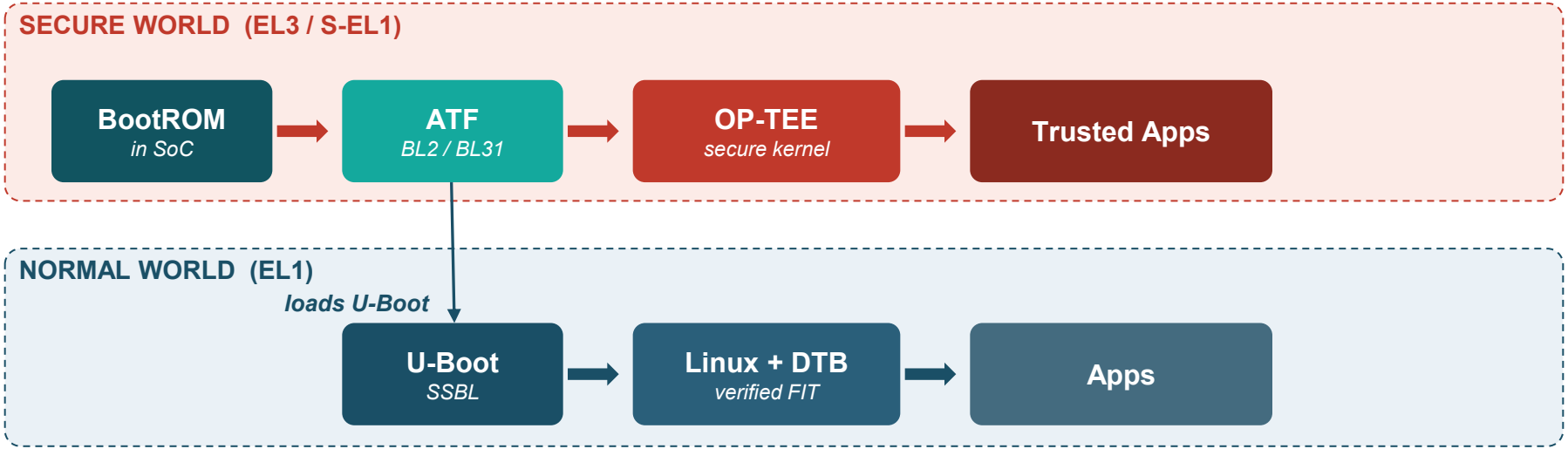
Isolates execution environments

ARM Trusted Firmware-A (TF-A)

- ARM's reference firmware for the early boot stages.
- Splits the work across **several boot levels (BLn)**, each with one clear job.



Secure boot → Secure OS



What is a Secure OS like OP-TEE?

OP-TEE:

Tiny OS for the Secure World
Open source, runs alongside Linux on the same CPU.

Hosts Trusted Applications
Drives secure peripherals: crypto engines, RNG, secure RAM.

Linux talks to it via a driver
OP-TEE client → SMC call → enter Secure World, return.

Isolated from Linux memory
A compromised kernel still cannot read OP-TEE's RAM.

⚠ **Don't confuse it with Secure Boot**

Secure Boot

When: at boot, once
Job: verify each stage before running it
Answers: is this code allowed to run?

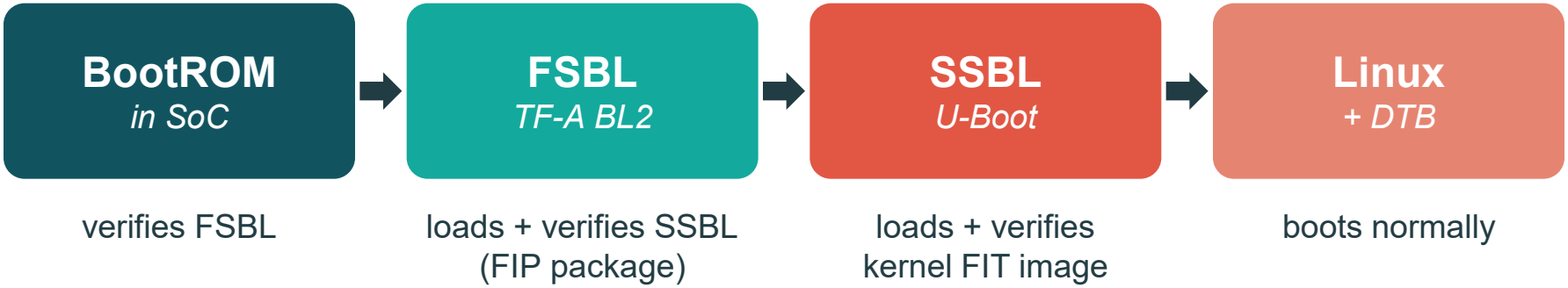
Secure OS

When: at runtime, always
Job: own secure peripherals and isolate sensitive code
Answers: who can touch this peripheral, now?

- **Secure Boot proves the code is yours.**
- **Secure OS guards the secure peripherals while it runs.**

Real example: STM32MP1 & 2

STMicroelectronics calls their Secure Boot **Trusted Boot Chain**:



Real example: NXP i.MX 9

NXP's name for their signed-image format is **AHAB** (Advanced High Assurance Boot), a container holding image, signature and key. Same idea as a FIT image.

AHAB container

NXP's signed-image format

One binary holding image, signature and SRK

SRK = Super Root Key, up to 4 of them

SRK hash burned in fuses at the factory

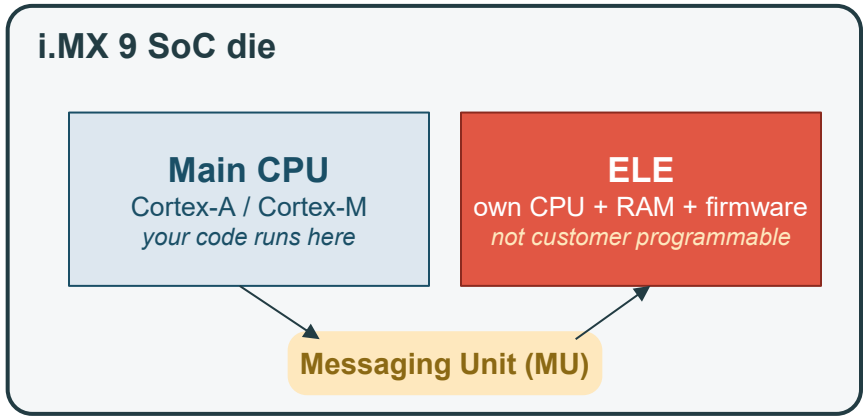
Boot flow on i.MX 9

ROM hands off to the security subsystem (ELE)

1. ROM hands the first AHAB container over
2. Signature checked vs SRK fuse hash
3. SPL and U-Boot reuse the same call
4. U-Boot verifies the FIT image

What is the EdgeLock Secure Enclave (ELE) ?

NXP's **ELE** is a small, isolated CPU on the same die that owns the keys and runs the verifier



Physically isolated
Own CPU core, own RAM, own NXP-signed firmware.

Owned by NXP
Firmware not customer programmable. You only call services.

Talk via Messaging Units
MU = mailbox the main CPU writes commands into.

Guards the keys
SRK fuses, OEM keys, RNG, AES engine live inside ELE.

Even if your Linux is fully compromised, the keys never leave ELE.

Conclusion

What Secure Boot does guarantee

✓ **The boot code is the one you signed**

Nobody swapped your bootloader or kernel for theirs.

✓ **The boot will refuse to run if anything is tampered**

Flip one byte in U-Boot, ROM stops. Flip one byte in the kernel, U-Boot stops.

✓ **You start every boot from a known state**

This is the foundation everything else builds on (TPM, attestation, dm-verity).

What Secure Boot does NOT guarantee

✗ That your code is bug-free

A signed kernel with a remote-code-execution bug is still vulnerable.

✗ That what runs after boot is safe

Userspace, packages, containers, network: none of that is covered.

✗ That your keys are safe

Leak the private signing key once, the entire fleet is compromised.

Common mistakes

- **Signing the kernel, forgetting the device tree**
  - Attacker patches the DTB cmdline to disable verity. Boot still passes.
- **Leaving U-Boot console open**
  - Attacker types one line, jumps to an unsigned address. Game over.
- **Not blowing the fuses**
  - SoC stays in "open" mode forever. Verification logic exists but is bypassed.
- **Keys on a developer laptop**
  - One stolen laptop = entire product compromised.
- **Testing only the happy path**
  - Flip a byte in your image. Does the board really refuse to boot? Verify it.

Thank you !

Roy Jamil

www.ac6-training.com

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference