Ch. 14 - IoT Security and Privacy

Sec 2 – Over-the-air updates

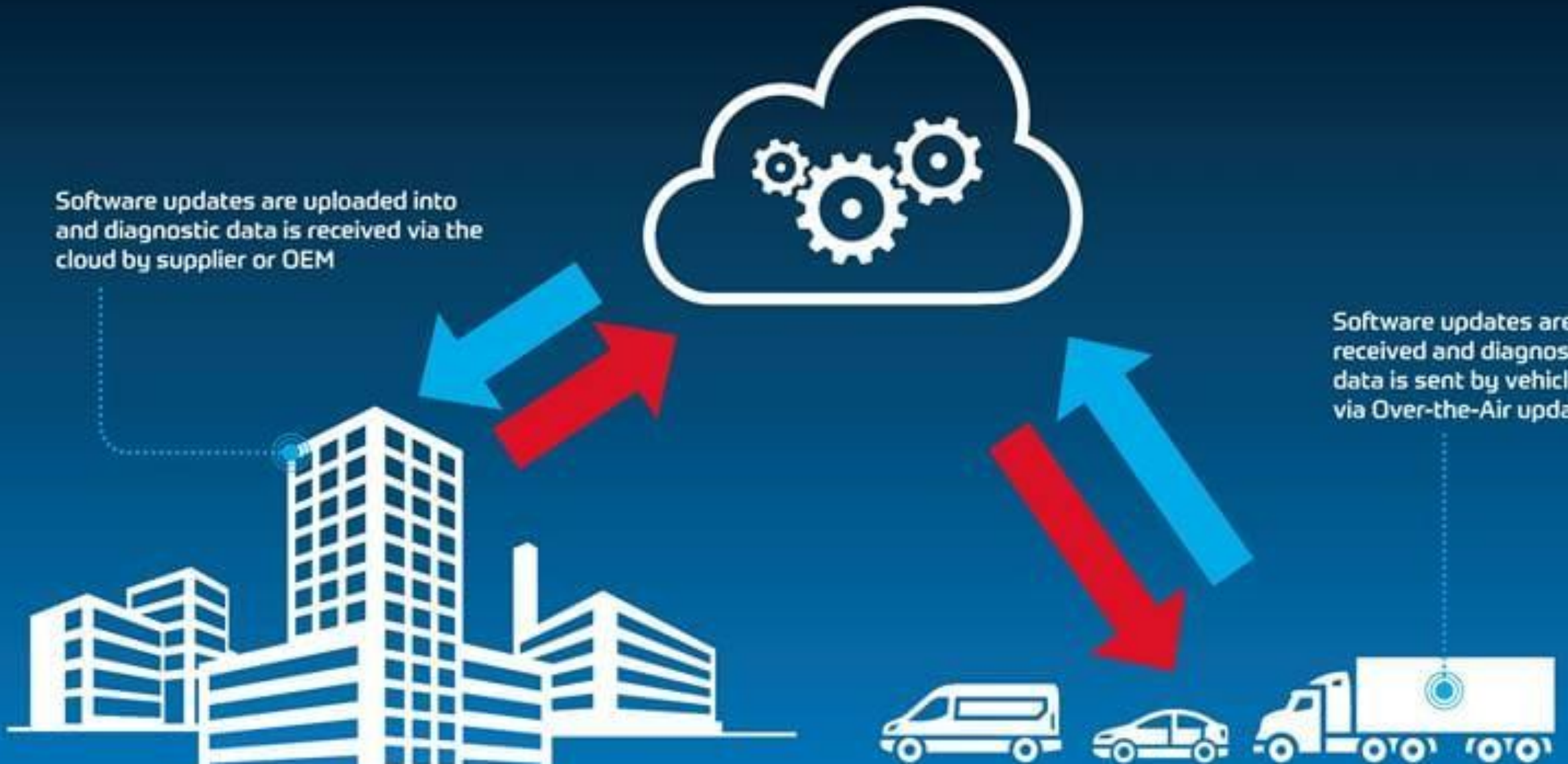
COMPSCI 244p

Internet-of-Things; Software and Systems

IoT Systems

- Must be extensible and flexible
- Once a potential vulnerability has been discovered, it need to be patched immediately.
- It must be quick and effortless

Over-the-Air Updates



Over-the-Air Updates

- Over-the-air means that a file is downloaded to the device having been sent from a cloud-based server across a Wi-Fi network or mobile network – either direct to the device, or to a gateway and then uploaded, for example through a Bluetooth connection.

- OTA is typically downloaded as a ‘delta’; rather than sending a full software installation file (a ‘full image’), the manufacturer sends only the part that needs to be changed. This can decrease the download time and reduces the manufacturer’s distribution costs.

- The combination of a software delta file and security credentials specific to the devices – these are crucial both onboard and offboard - is called an ‘update package’. The package may contain several delta files, to update different modules on the vehicle. OTA must be hack-proof – device manufacturers need to ensure that only appropriate updates get applied to their vehicles.

- Very common in cars.
  Updates that can be sent this way include changes to the software that controls the vehicle’s physical parts, its electronic signal processing system, or user interfaces such as infotainment screens and instrument clusters.

Client Server Architecture is required

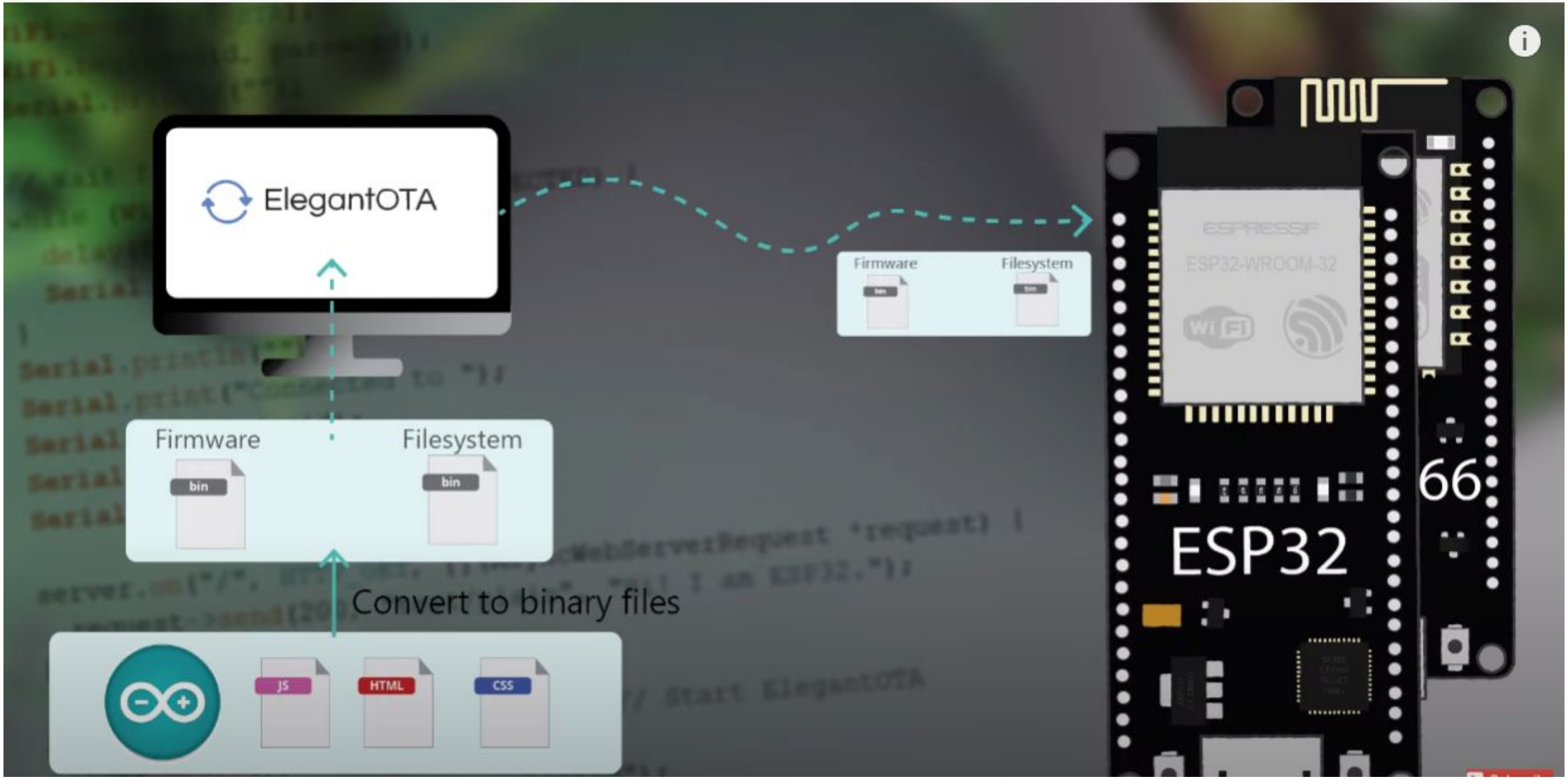
- Requires management of users
- Scalable, highly-available cluster
- User friendly interface
- Multi-level Security (Authentication+Authorization)
- API supports for clients to talk to
- Adaptive Delta compression
- Provides Insights/Analytics

Server

- Agnostic to hardware and CPU
- Supports rollback and recovery
- Lightweight to fit in the flash
- Support self-update and external updates
- Integrates into security framework

Client

How to implement OTA with ESP32



OTA with internal library

- ESP32 Arduino comes with in-built OTA support

- Allows you to use Wi-Fi instead of Serial to upload firmware and open Serial port

```
[env:myenv]
platform = espressif32
board = esp32dev
framework = arduino
upload_protocol = espota
upload_port = 192.168.0.255
```

- References

- https://docs.platformio.org/en/latest/platforms/espressif32.html#over-the-air-ota-update
- https://github.com/espressif/arduino-esp32/blob/master/libraries/ArduinoOTA/examples/BasicOTA/BasicOTA.ino

OTA with external library

https://github.com/ayushsharma82/AsyncElegantOTA

- Step 1: Upload firmware
- Step 2: Update file system (SPIFFS)

- What's the catch: These libraries do not scale by default

More security features present in ESP32

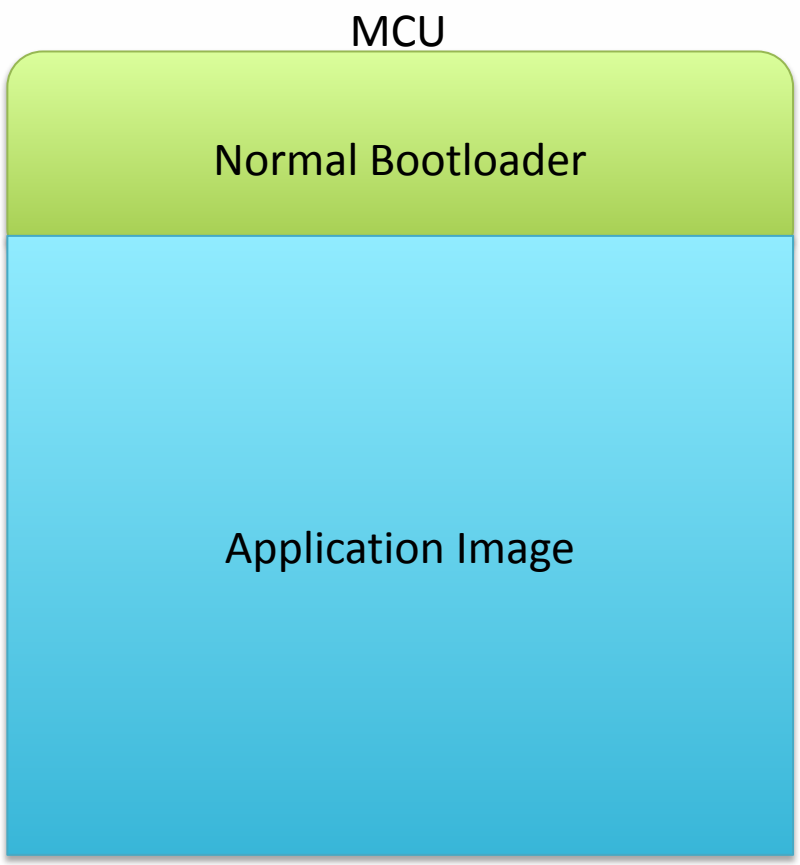
- **Secure Boot** protects a device from running any unauthorized (i.e., unsigned) code by checking that each piece of software that is being booted is signed.
  - On a device, these pieces of software include the second stage bootloader and each application binary.
  - Note that the first stage bootloader does not require signing as it is ROM code thus cannot be changed.

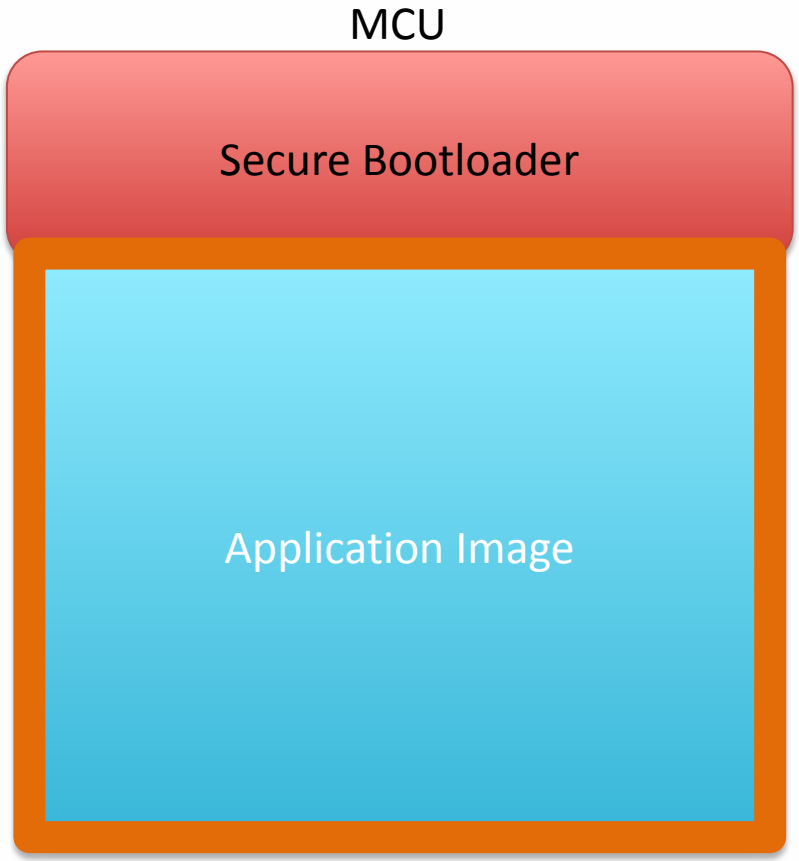
- **Flash encryption** is intended for encrypting the contents of the device's off-chip flash memory.
  - Once this feature is enabled, firmware is flashed as plaintext, and then the data is encrypted in place on the first boot.
  - As a result, physical readout of flash will not be sufficient to recover most flash contents.

eFUSE: One Time Programmable

- ESP32 has a 1024-bit eFUSE, which is a one-time programmable memory.
- This eFUSE is divided into 4 blocks of 256-bits each.
- Blocks 1,2 store keys for flash encryption and secure boot respectively.
- Once the keys are stored in the eFUSE, it can be configured such that any software running on ESP32 cannot read (or update) these keys (**disable software readout**).
- Once enabled, only the ESP32 hardware can read and use these keys for ensuring secure boot and flash encryption.

Standard vs Secure Bootloaders





Authentication + Integrity

How secure boot works: Step 1 – Generate keys

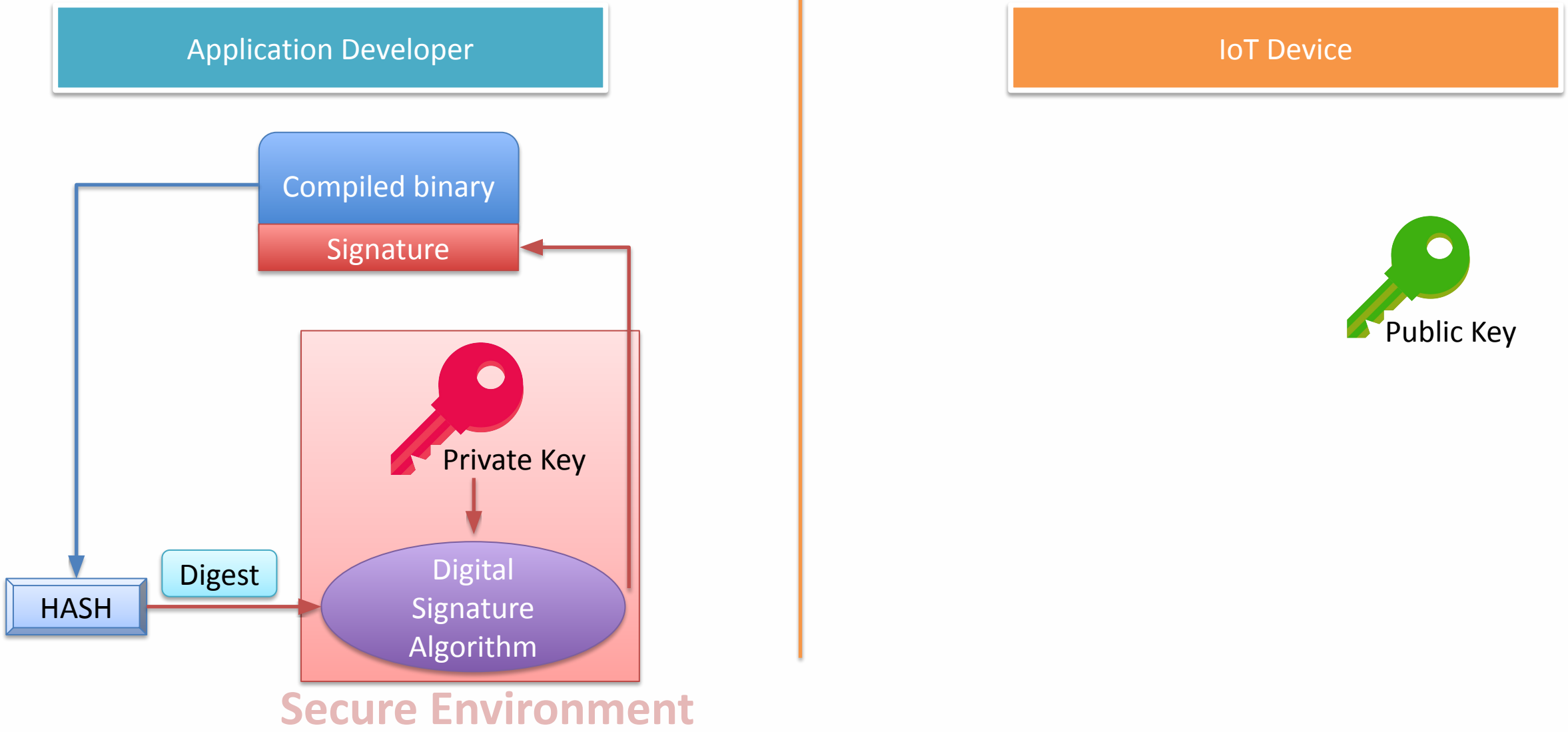
Application Developer

IoT Device

Private Key

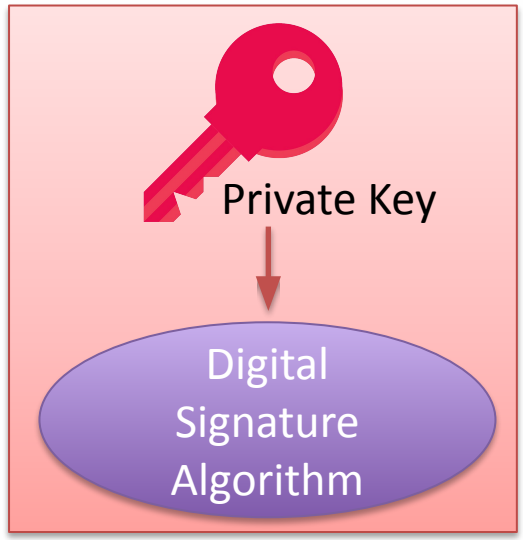
Public Key

How secure boot works: Step 2 – Digitally Sign firmware



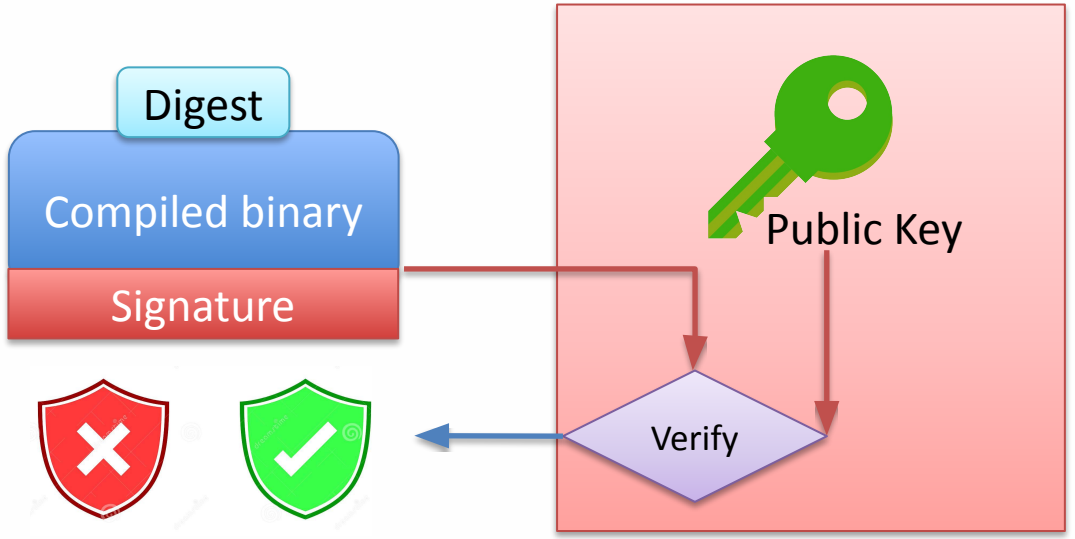
How secure boot works: Step 3 – Verify signature

Application Developer



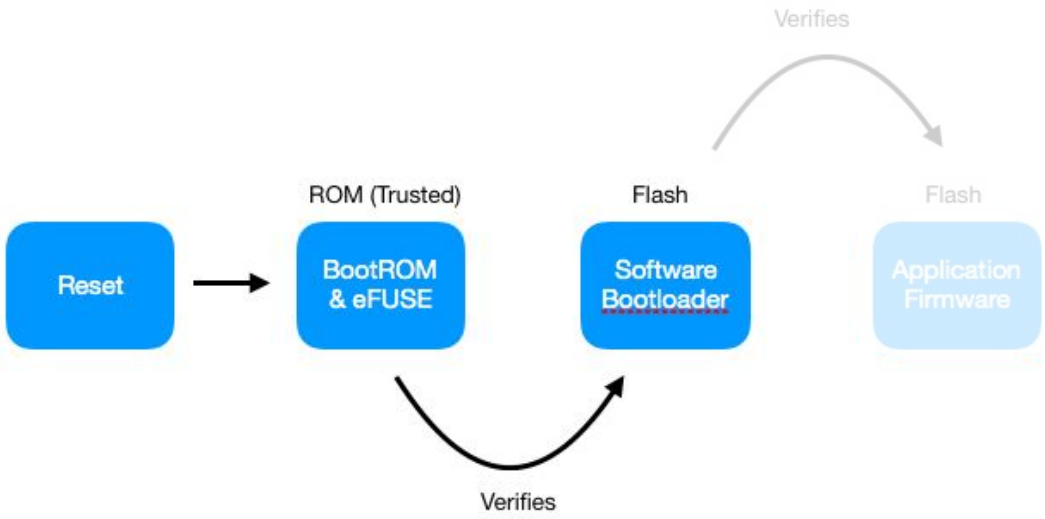
Secure Environment

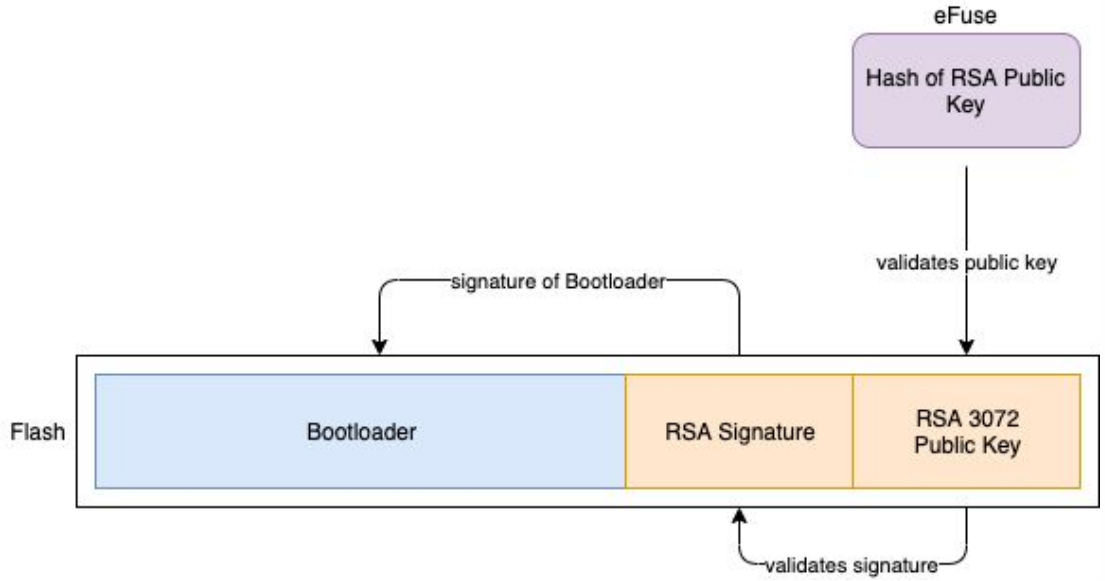
IoT Device



Secure Environment

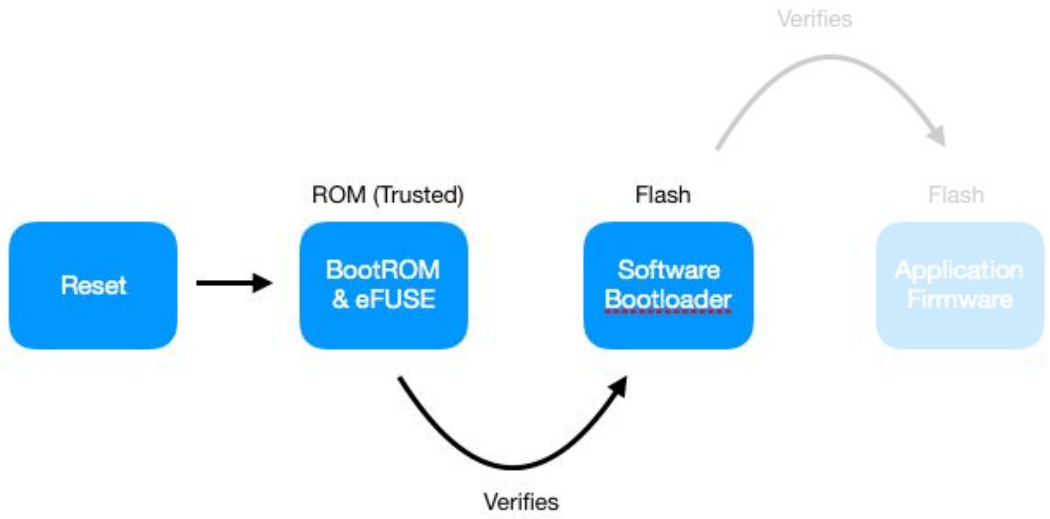
Secure boot: Chain of trust

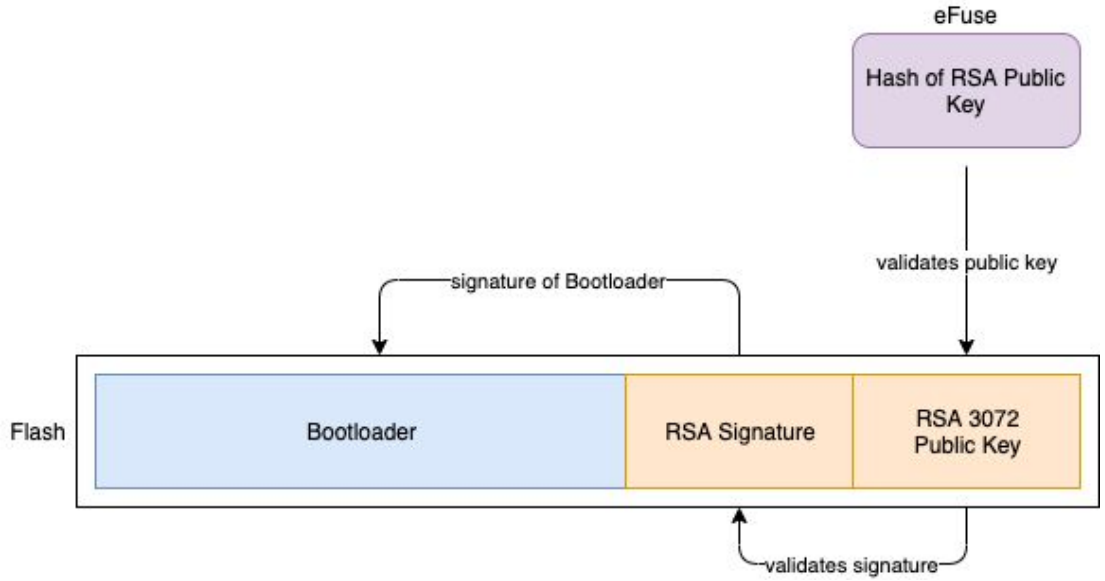




- On reset, the BootROM begins execution. This is in the ROM, within the hardware, and hence is trusted.
- The eFuse, again in the hardware (with software read/write protection), is trusted.
- The BootROM looks up the RSA3072 Public Key from the Bootloader's image and validates its checksum with the one stored in the eFuse. This validates the RSA3072 Public Key.
- Now the RSA Public Key is used to validate the bootloader itself using RSA sign-verify mechanism.

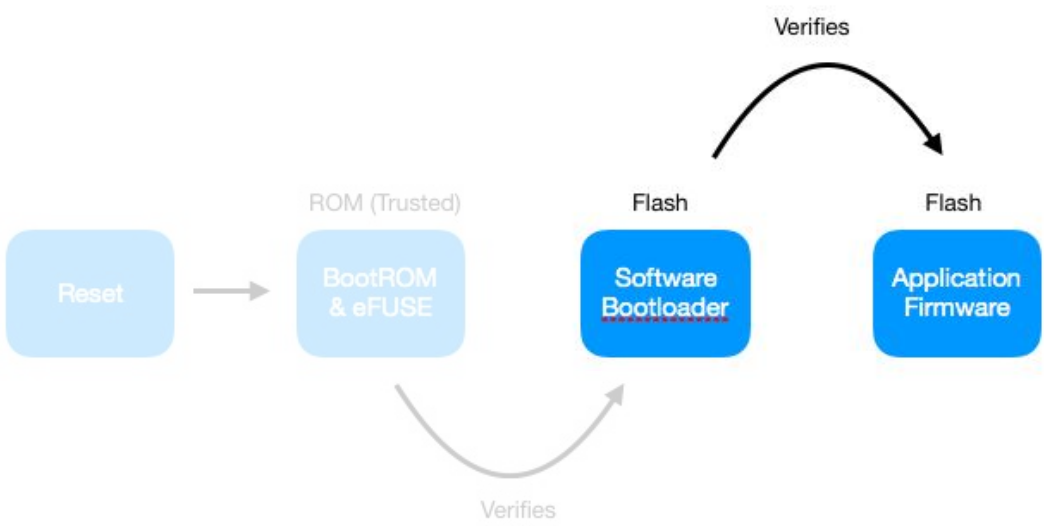
Secure boot: Chain of trust

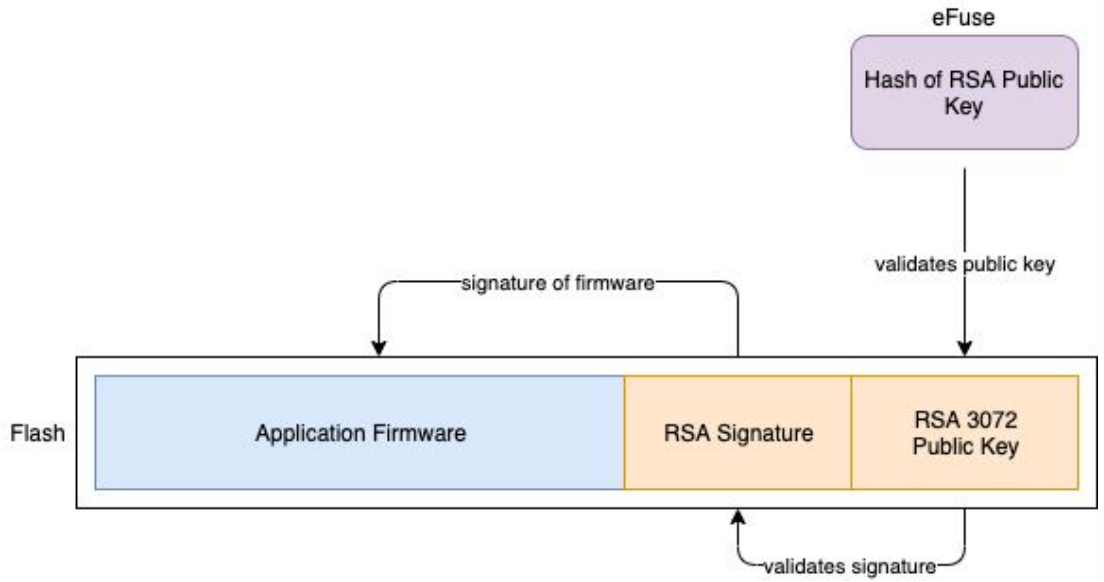




Q: Why store the Hash of the public key in eFuse instead of public key directly ?

Secure boot: Chain of trust

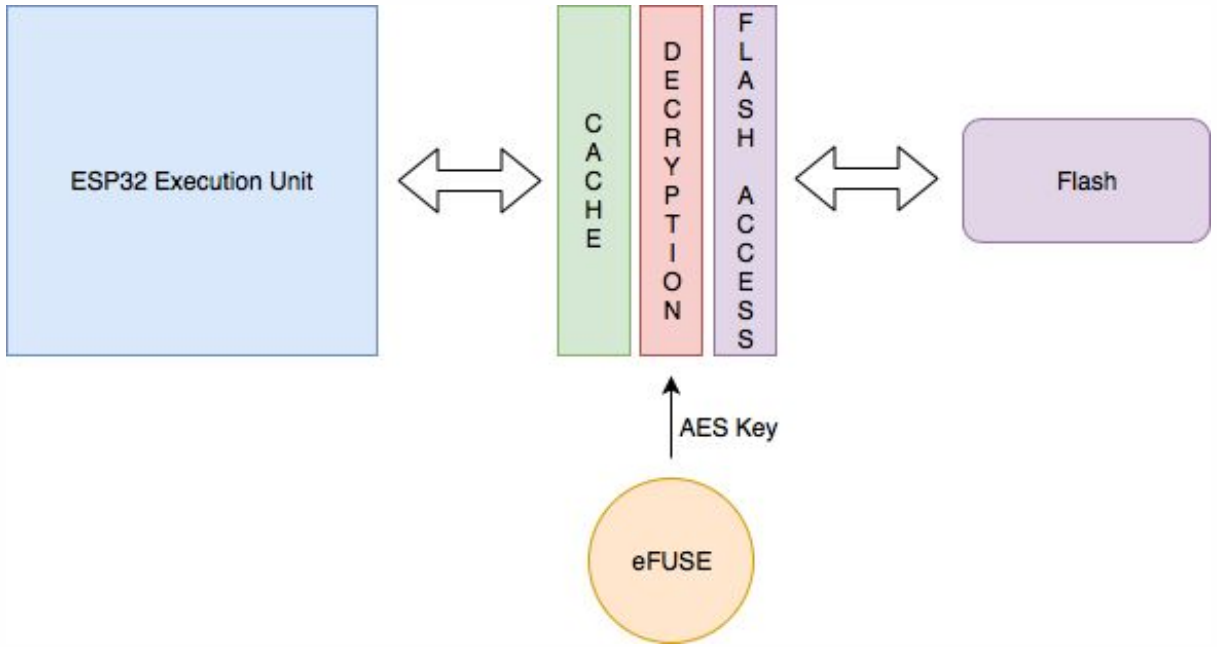




- Once Bootloader is trusted, the BootROM transfers control to Bootloader.
- Bootloader then repeats the process to validate application firmware (same process)
- Finally, ESP32 boots securely !

Flash encryption

- When flash encryption is enabled, all memory-mapped read accesses to flash are transparently, and at-runtime, decrypted.
- The flash controller uses the AES key stored in the eFUSE to perform the AES decryption.
- Similarly, any memory-mapped write operation causes the corresponding data to be transparently encrypted before being written to flash.



What about UART/JTAG

- The eFuse has one-time programmable bit fields that allow you to disable support for JTAG debugging, as well as the support for UART Boot.
- Once disabled, these features cannot be re-enabled on the device under consideration.

Questions