# Device Security: Trusted boot and code signing

Mihai Ordean
Designing Secure Systems
University of Birmingham

# Overview

- **Device security**
  - Is code on the device vulnerable to exploits ? (e.g. buffer overflows)
  - Is the code authenticated ? (i.e. has not been tampered with)
- Data security
  - Is the stored data is accessible to everyone? (e.g. encrypted)
  - Is the stored data authenticated?
- Metadata security
  - What does metadata reveal about data?
  - Can we tamper the metadata?
- Protocol security
  - Is data in transit visible?
  - Can data in transit be tampered with?

# Overview

- Device security
  - Physical device security (e.g. side-channel attacks vulnerabilities)
  - **Firmware/OS security  (e.g. bootloader, kernel)**
  - **Application security (e.g.  code signing, sandboxing)**

# Introduction

# Security challenges

Protect devices against:

- Malicious applications
- Rootkits

# Malicious applications

Malware distributed using OS specific applications, designed to exploit the operating system vulnerability's.

## Issues:

- High success rate because they masquerade as useful applications
- Are often used as a means to install more dangerous malware: backdoors, rootkits

# Rootkits

Code designed to enable access to a computer or areas of its software that would not otherwise be allowed.

## Issues:

- Install themselves with highest privileges
- Prevent detection/removal with anti-malware tools

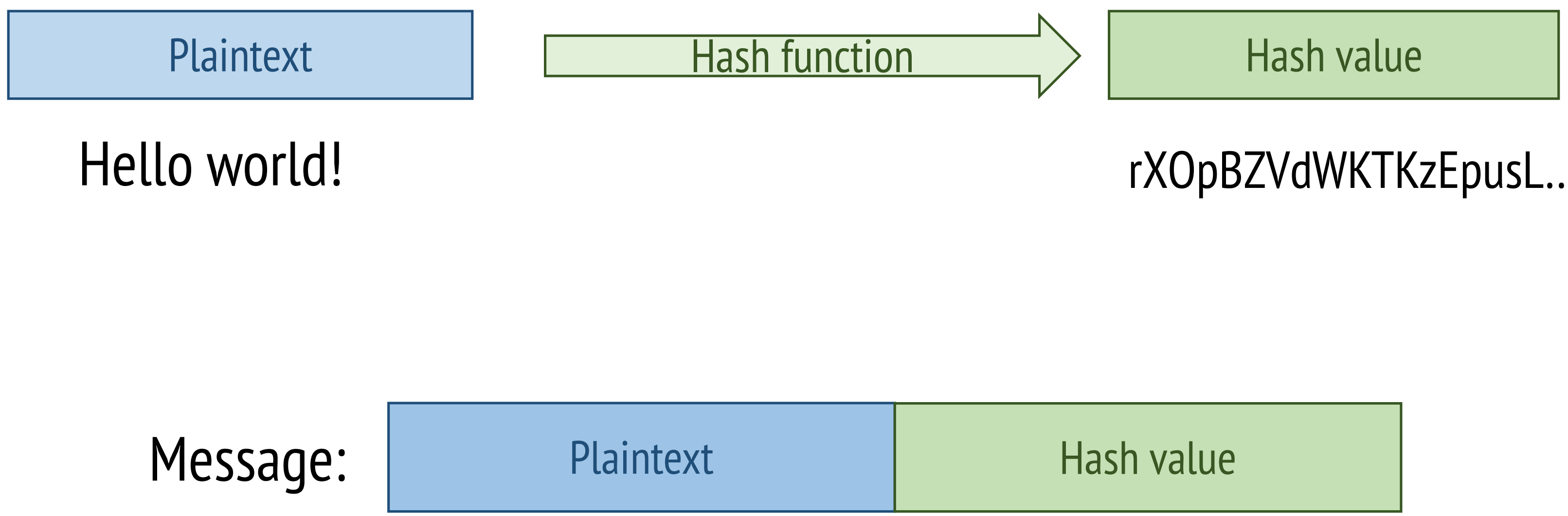
# Mitigations

Defence in depth:

- Secure the boot process
- Secure the user space
- Secure the distribution channels

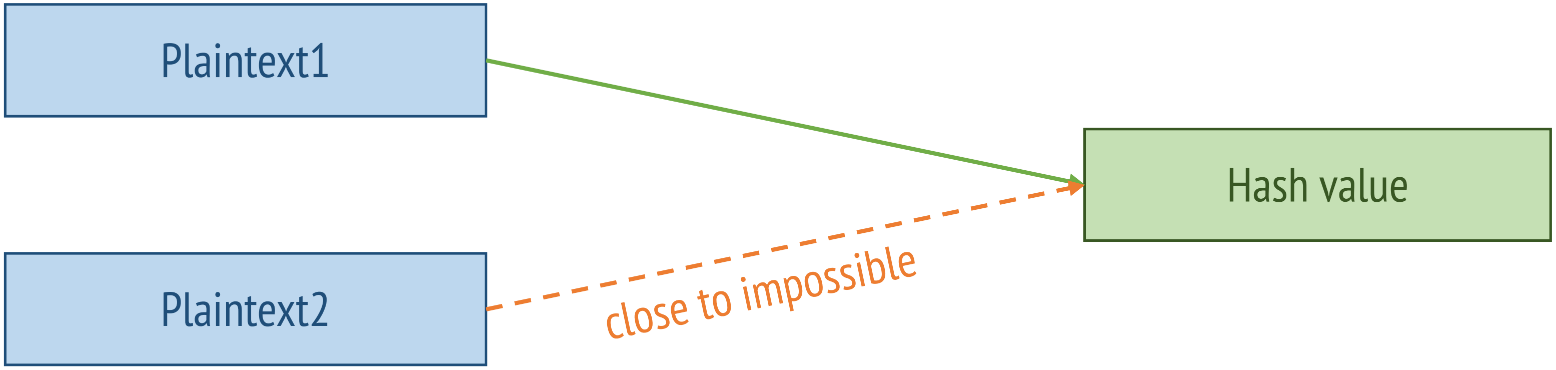
# Hash functions and signatures

# Hash functions



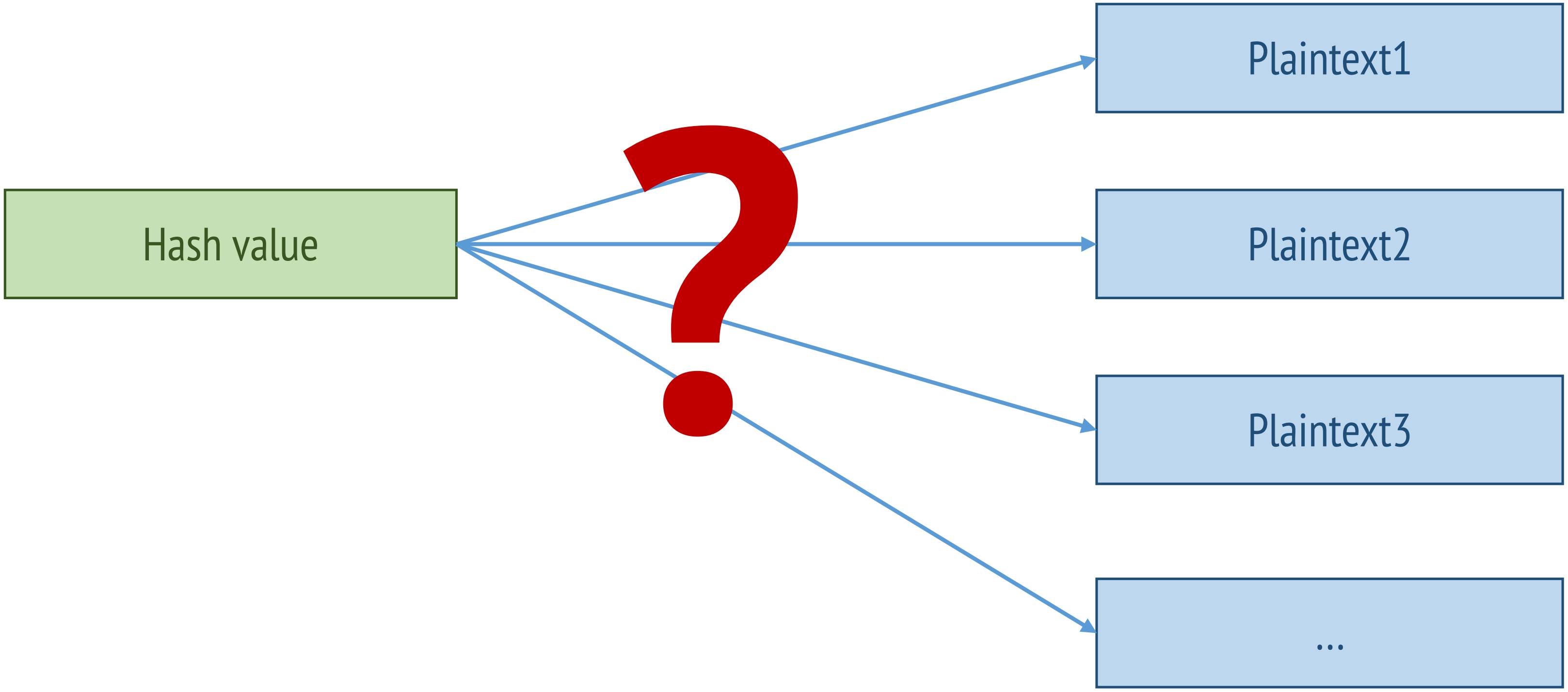
# Hash functions

Unique per plaintext (low collision)



# Hash functions

Difficult to reverse (one way)



# Hash functions

Size of Hash value is independent from size of Plaintext .

Size of Hash value is given by the **hash function**.

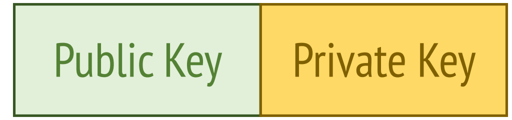
# Hash functions

Size of Hash value is independent from size of Plaintext .

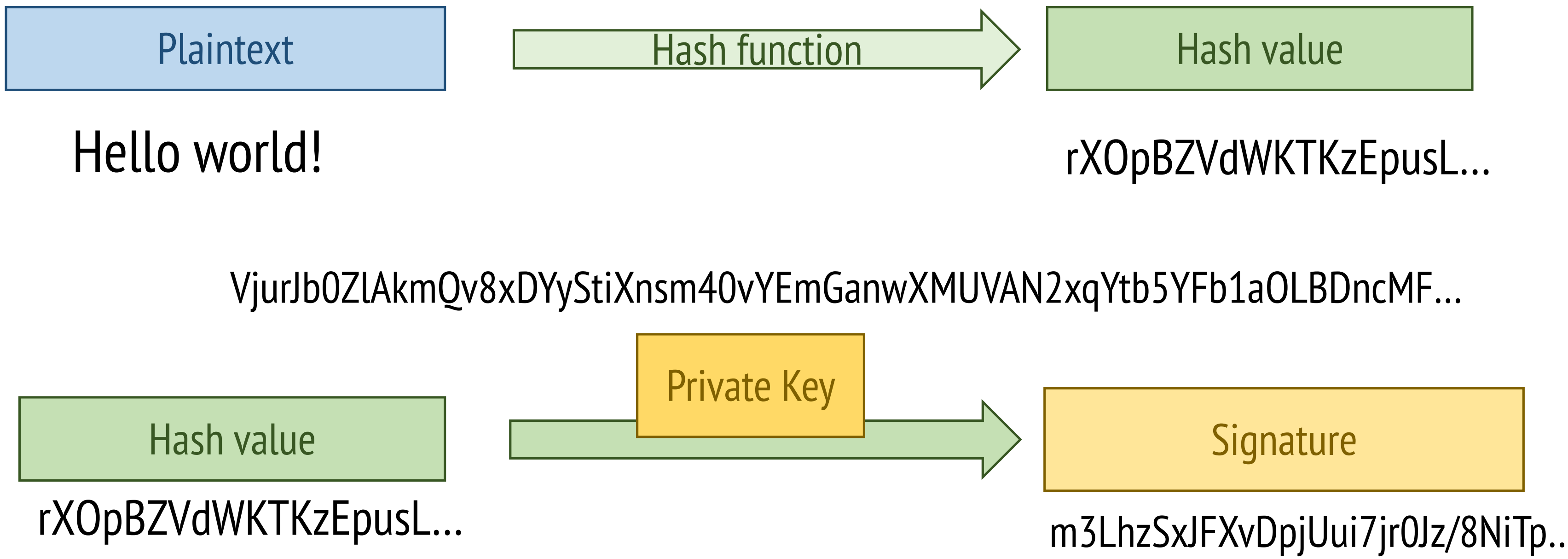
Size of Hash value is given by the **hash function**.

Example:
SHA256 = 256 bits
SHA512 = 512 bits

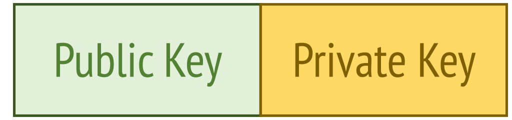
# Signatures



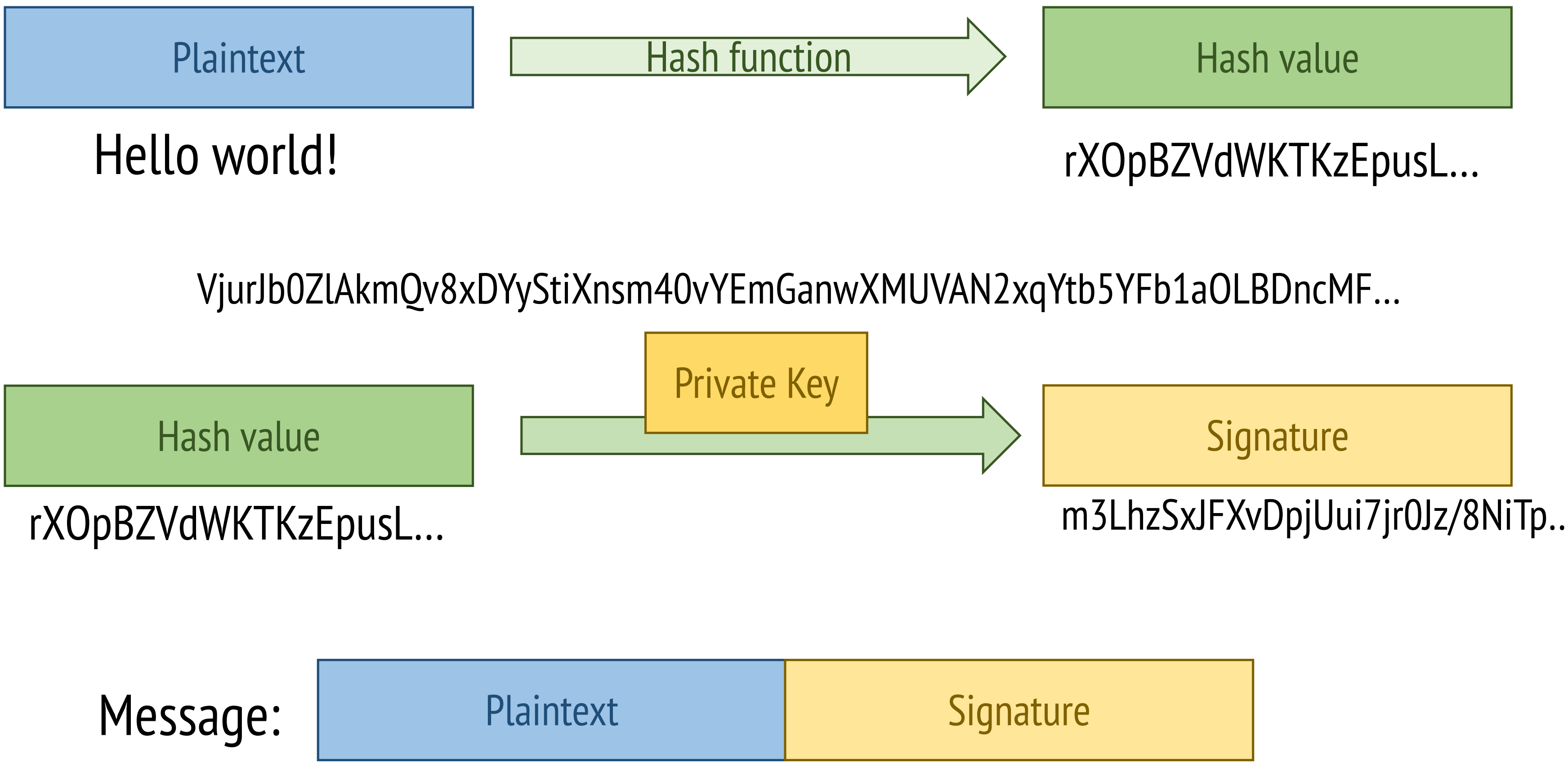

## Signing:



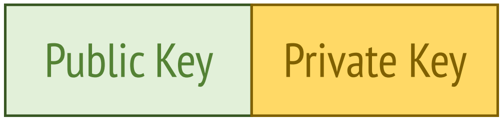
# Signatures



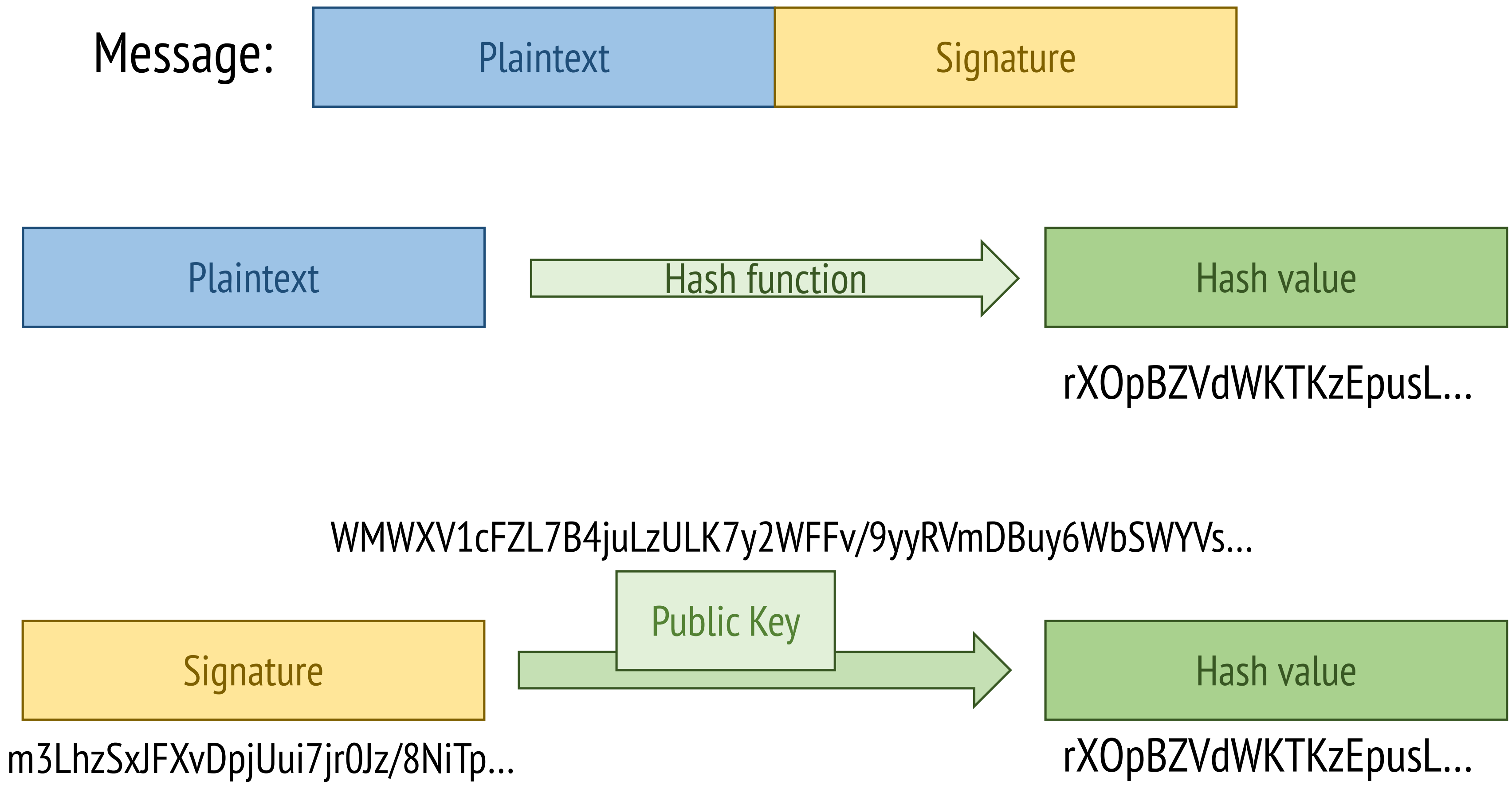

## Signing:



# Signatures

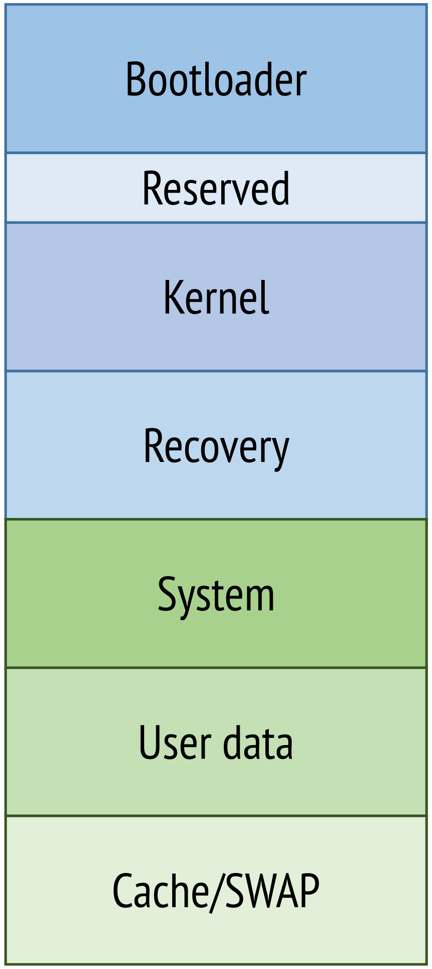



## Verification:

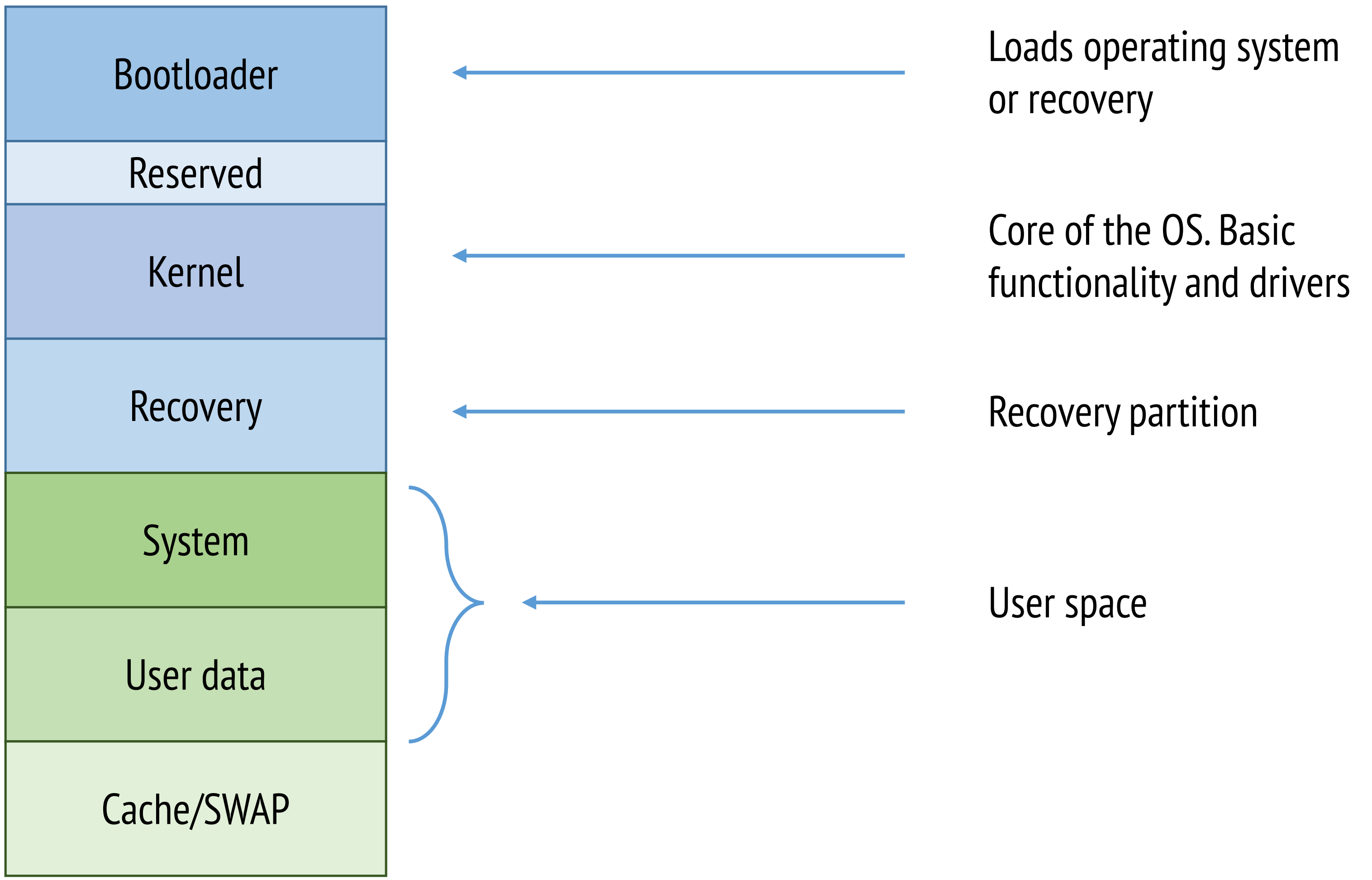


# System architectures

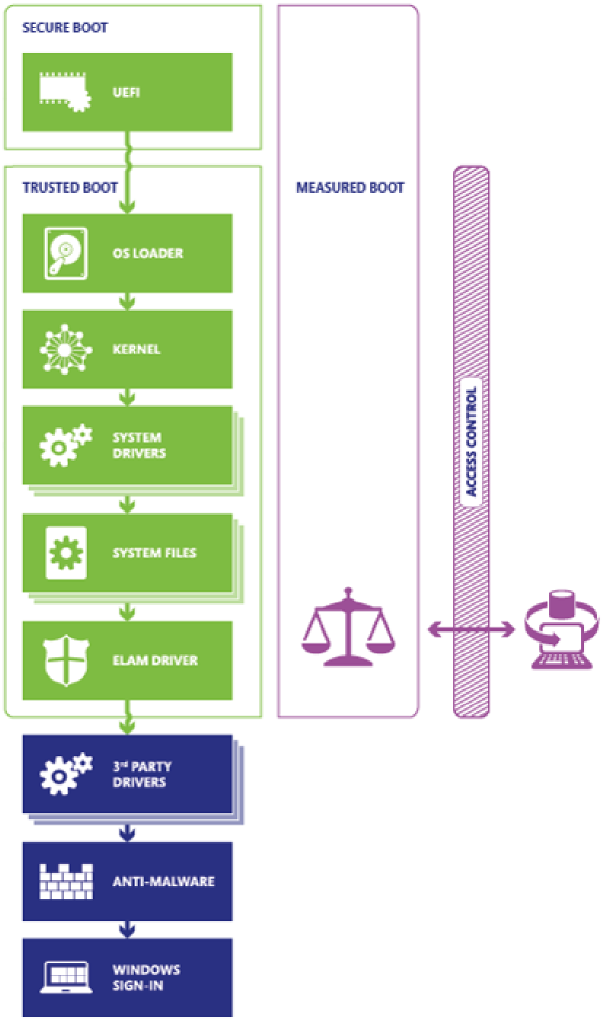
# Generic architecture

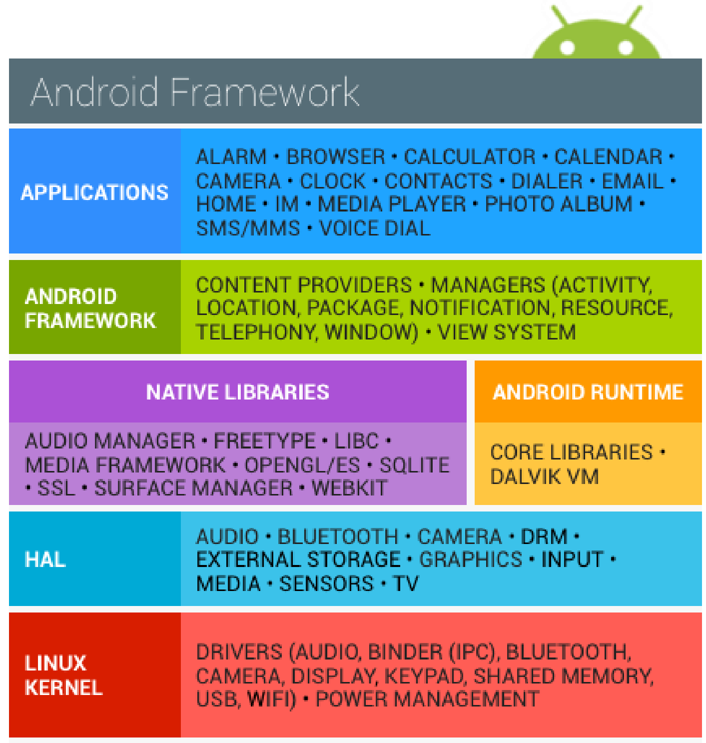


# Generic architecture

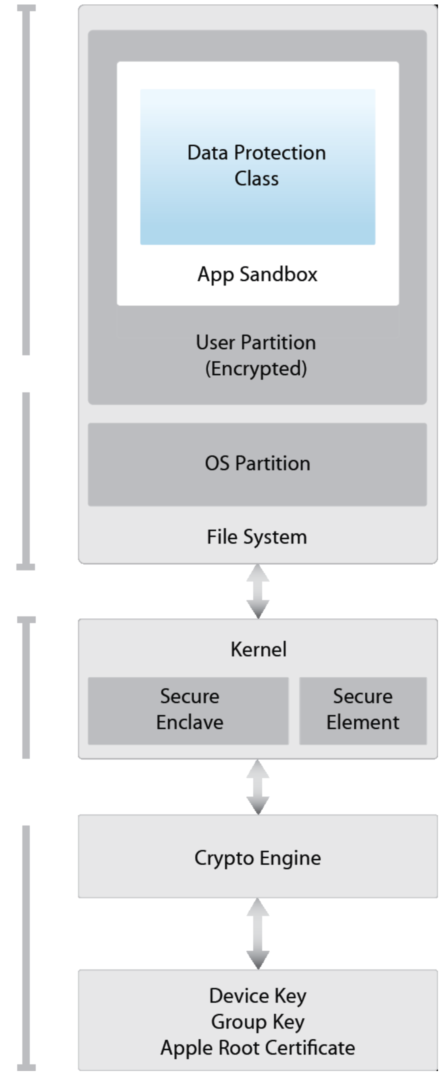


# System architecture



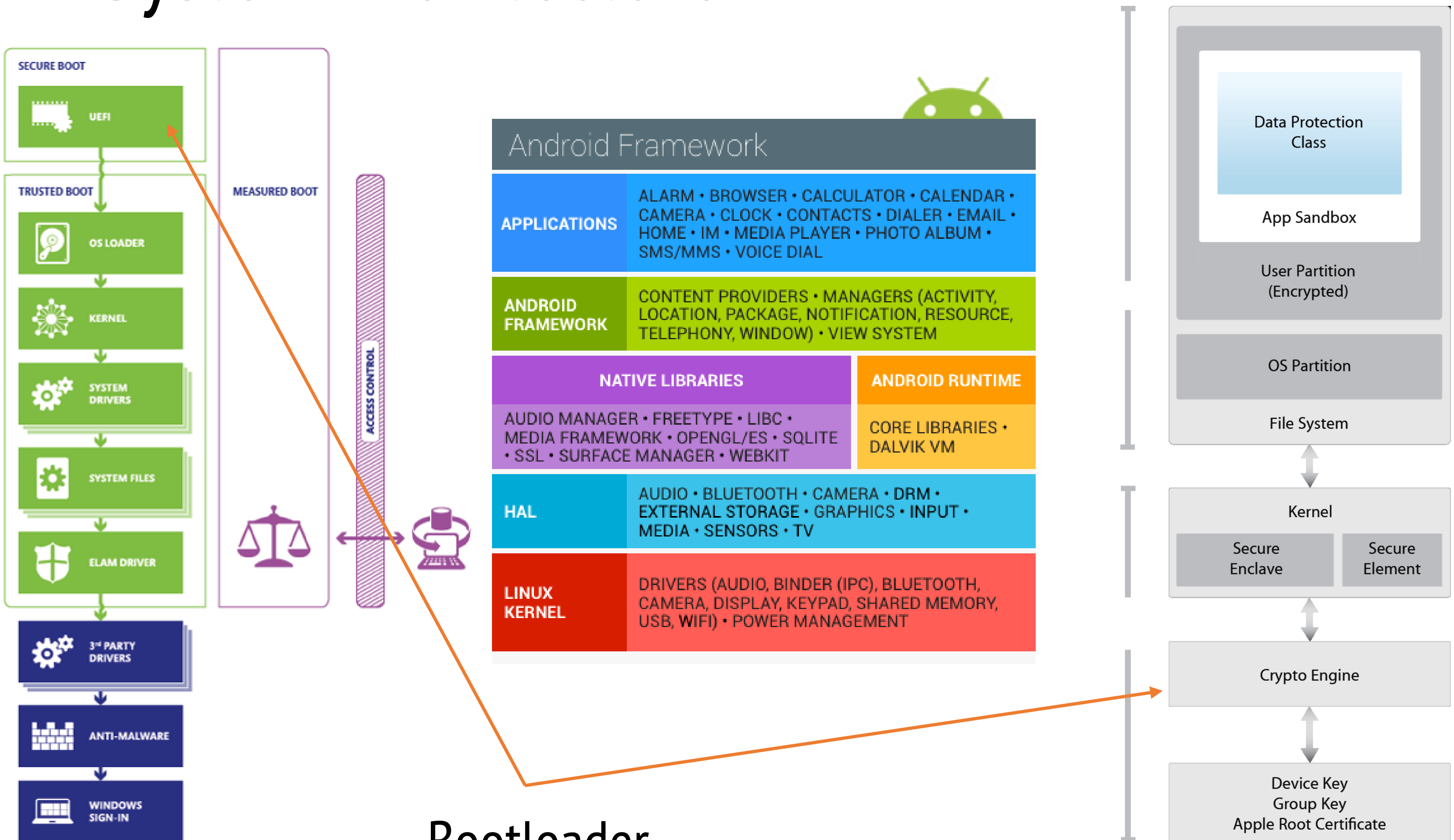

Android architecture




Windows architecture

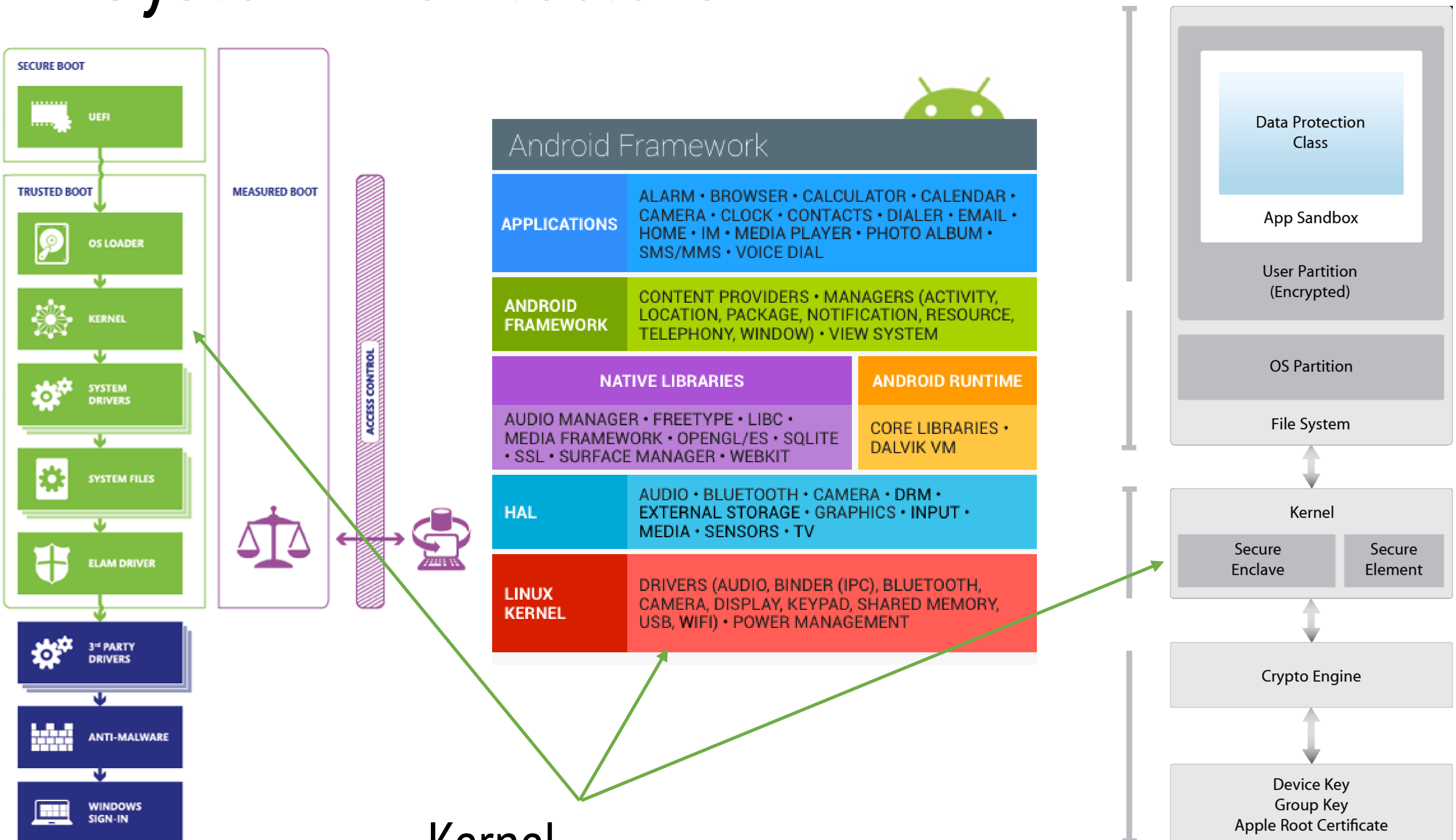
IOS security architecture

# System architecture




Bootloader

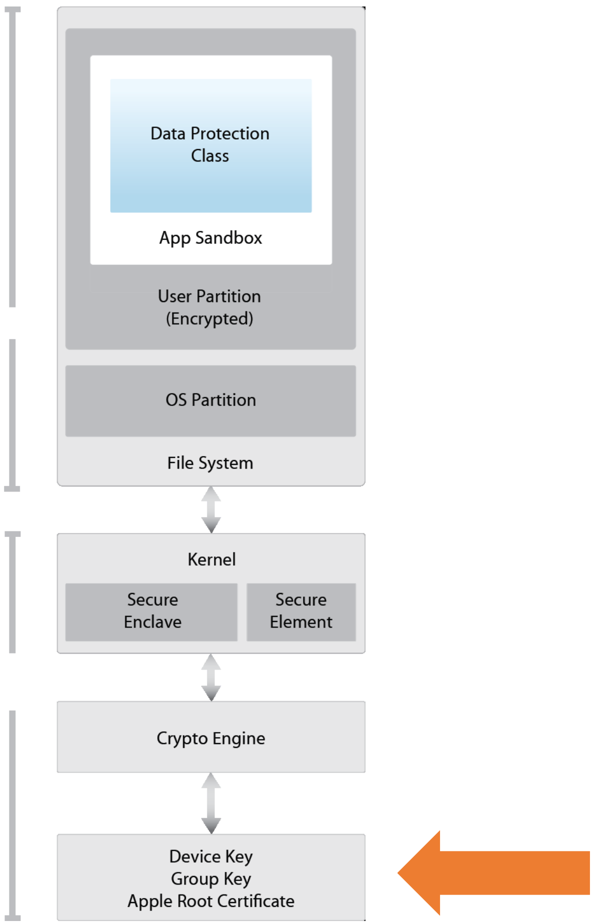
# System architecture




Kernel

# Trusted boot

# Root of trust




1. The bootloader is the guardian of the device state and is responsible for initializing the TEE and binding its root of trust.
2. If rooting software compromises the system before the kernel comes up, it will retain that access.
3. The bootloader verifies the integrity of the boot and/or recovery partition before moving execution to the kernel.
4. **Hardware** root of trust is fixed because is laid down during chip fabrication.
5. **Non-hardware** root of trust can be changed because is stored on non-volatile memory (e.g. NAND).

# How to verify?

# How to verify?

Directly hash its contents and compare them to a stored value.

# How to verify?

Directly hash its contents and compare them to a stored value.

Some issues:

- Verifying an entire block device can take an extended period
- Will consume much of a device's power

# Dm-verity (android)

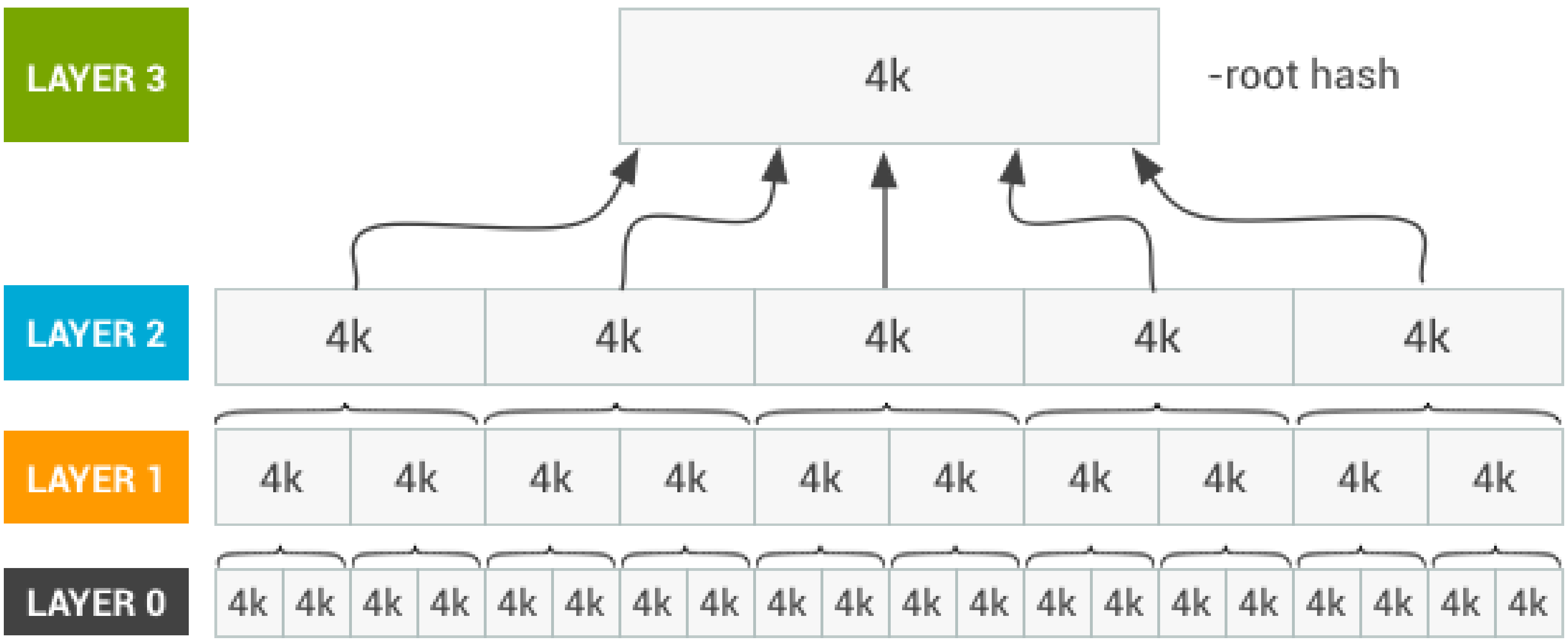
Uses:

- Block storage
- Cryptographic hash function, e.g. SHA256
- Cryptographic hash tree (i.e. Merkel Tree)

# Benefits of Dm-verity

- Verifies blocks individually and only when each one is accessed
- The HASH operation is done when the block is read into memory: the block is hashed in parallel
- The hash is then verified up the tree

# Boot/recovery partition




Android partition verification

# Question

- What properties does the root hash provide?

# Question

- What properties does the root hash provide?
  - Integrity

# Question

- What properties does the root hash provide?
  - Integrity
- How can we provide authentication to the data?

# Question

- What properties does the root hash provide?
  - Integrity
- How can we provide authentication to the data?
  - Use a key to sign the root hash

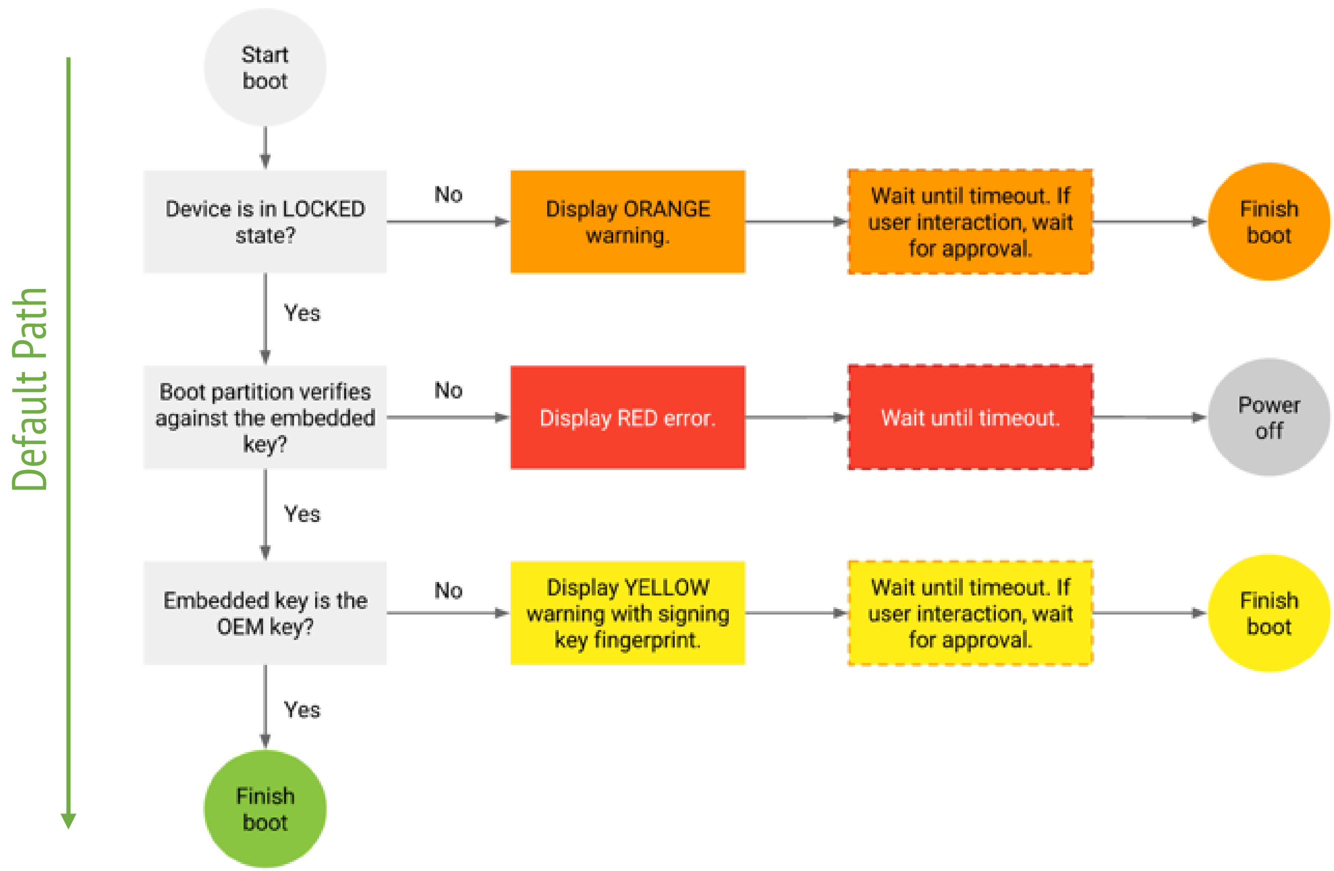
# Question

- What properties does the root hash provide?
  - Integrity
- How can we provide authentication to the data?
  - Use a key to sign the root hash
- Can this method be applied to all types of partitions? Why?

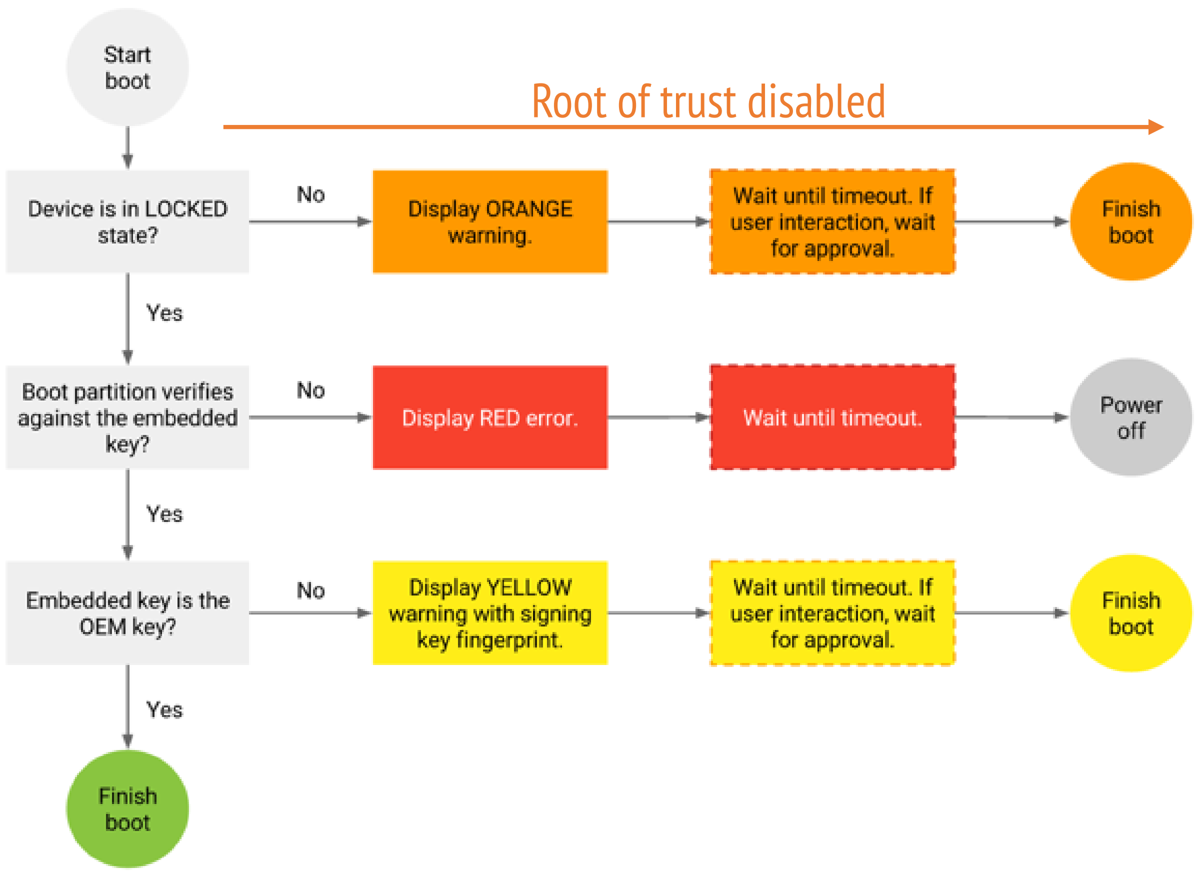
# Question

- What properties does the root hash provide?
  - Integrity
- How can we provide authentication to the data?
  - Use a key to sign the root hash
- Can this method be applied to all types of partitions? Why?
  - Just to read-only. Read-write partitions' HASH values would change whenever data is modified.

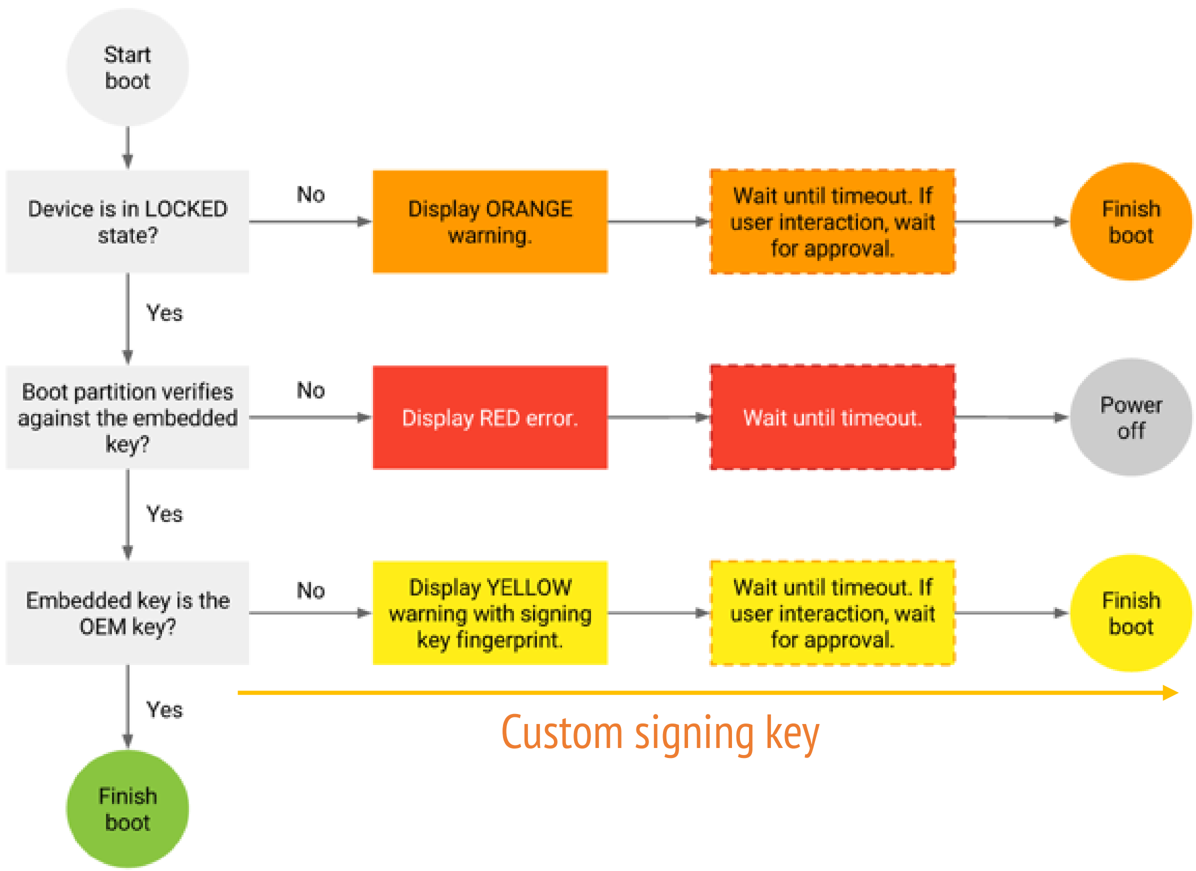
# Boot flow (Android)



# Boot flow (Android)



# Boot flow (Android)



# Main principles applied

- Authenticate: every component of the system is authenticated
- Control access: prevent loading the OS without a secure boot loader
- Make security usable: simple messages aimed at users
- Fail secure: stop the booting process if anything goes “wrong”
- …

# Exercise

Describe necessary steps to perform a firmware update

# Exercise

Describe necessary steps to perform a firmware update

1. Boot in a secure/trusted mode

# Exercise

Describe necessary steps to perform a firmware update

1. Boot in a secure/trusted mode
2. Check the signature of the firmware update using trusted ROM key

# Exercise

Describe necessary steps to perform a firmware update

1. Boot in a secure/trusted mode
2. Check the signature of the firmware update using trusted ROM key
3. Install firmware update

# Exercise

Describe necessary steps to perform a firmware update

1. Boot in a secure/trusted mode
2. Check the signature of the firmware update using trusted ROM key
3. Install firmware update
4. Update the bootloader to match the new firmware

# Exercise

Describe necessary steps to perform a firmware update

1. Boot in a secure/trusted mode
2. Check the signature of the firmware update using trusted ROM key
3. Install firmware update
4. Update the bootloader to match the new firmware

**Any thing else?**

# Individual research

Well…

Firmware downgrade attacks: installing an older firmware which might have (known) vulnerabilities.

**Give some solutions to prevent this!**

# What else?

- Kernel secure?
    - Yes
- Can we update OEM keys in case of compromise?
    - No

# What else?

- Kernel secure?
    - Yes
- Can we update OEM keys in case of compromise?
    - No
- What if we can still compromise the system by exploiting some design flaw(s) ?
    - Can we at least detect that and notify the user? (we could before)
    - **Can we prevent sensitive data compromise? (e.g. encryption keys)**

# Secure the boot process using hardware

# Hardware anchored security

## Trusted Platform Module (TPM)

- a secure cryptoprocessor
- generates cryptographic material (keys and random numbers)
- performs remote attestation
- protects cryptographic material by binding and sealing

## Trusted Execution Environment (TEE)

- secure area of the main processor
  (e.g. TrustZone in ARM, SGX in Intel)
- code and data loaded inside are protected with respect to confidentiality and integrity
- provides isolated execution

# What can I do with a TPM?

- Platform integrity
- Disk encryption
- Password protection
- Digital rights management
- Protection and enforcement of software licenses
- Prevention of cheating in online games

# What can I do with a TPM?

- **Platform integrity**
- Disk encryption
- Password protection
- Digital rights management
- Protection and enforcement of software licenses
- Prevention of cheating in online games

# Platform integrity

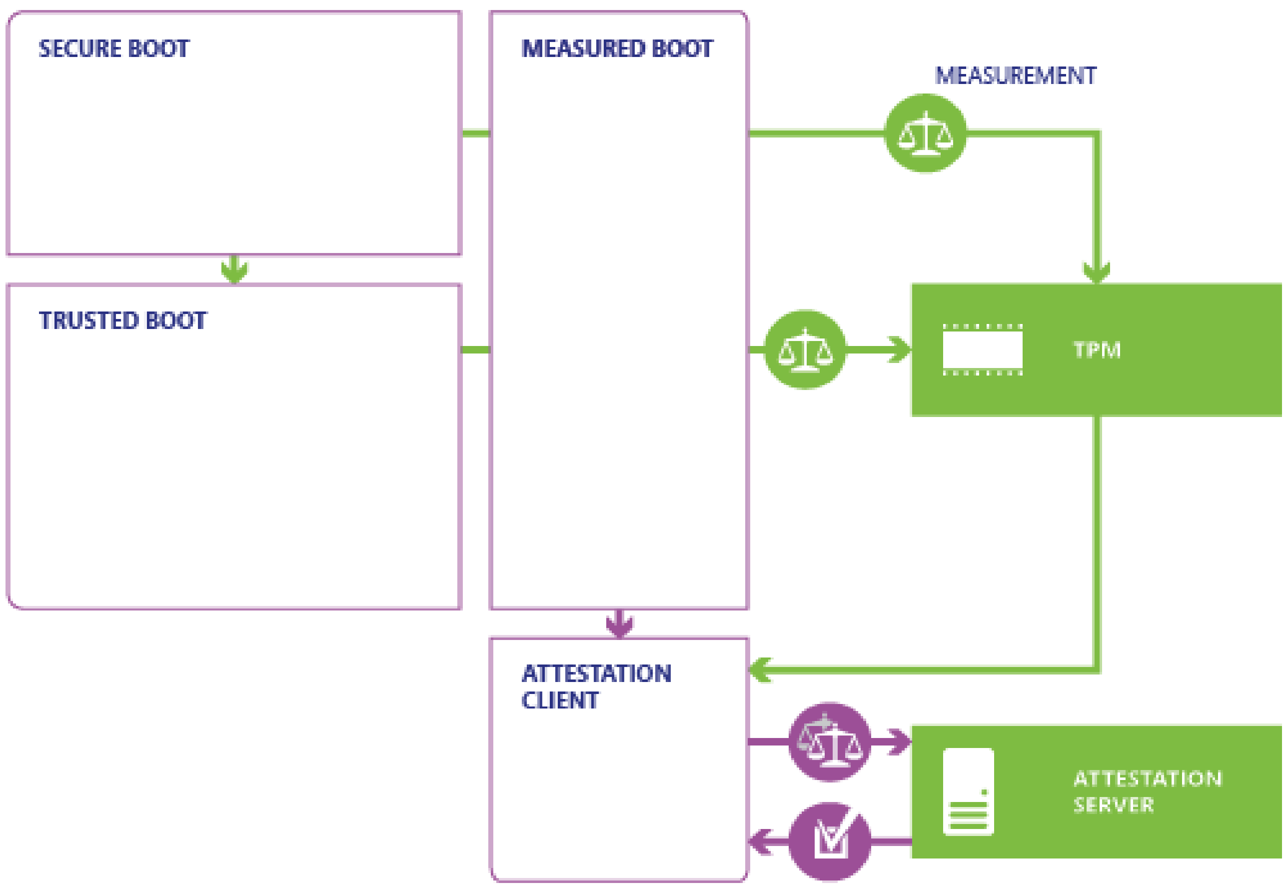
Requirements

- TPM
- UEFI
- Linux Unified Key Setup (LUKS) or BitLocker Drive Encryption

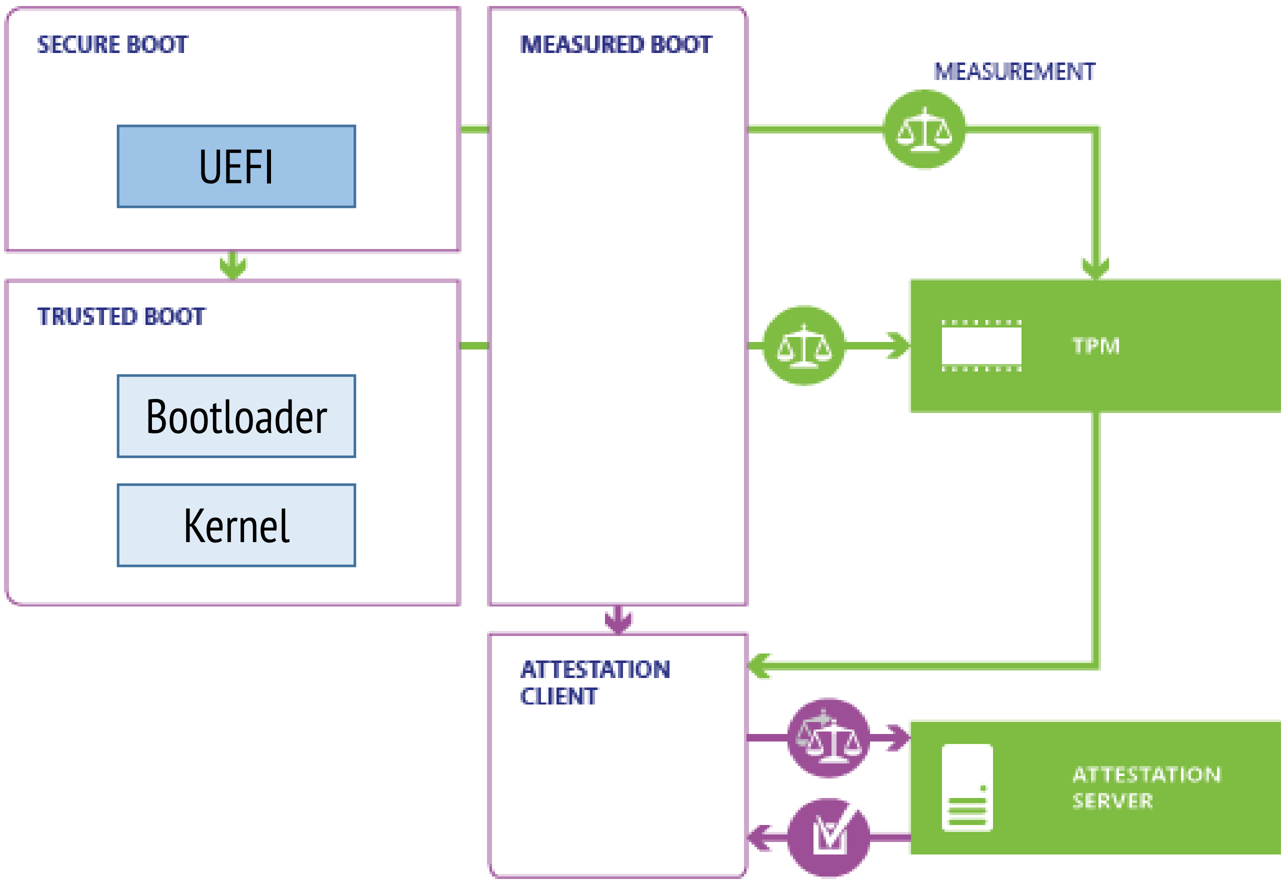
Enforces

- "root of trust"

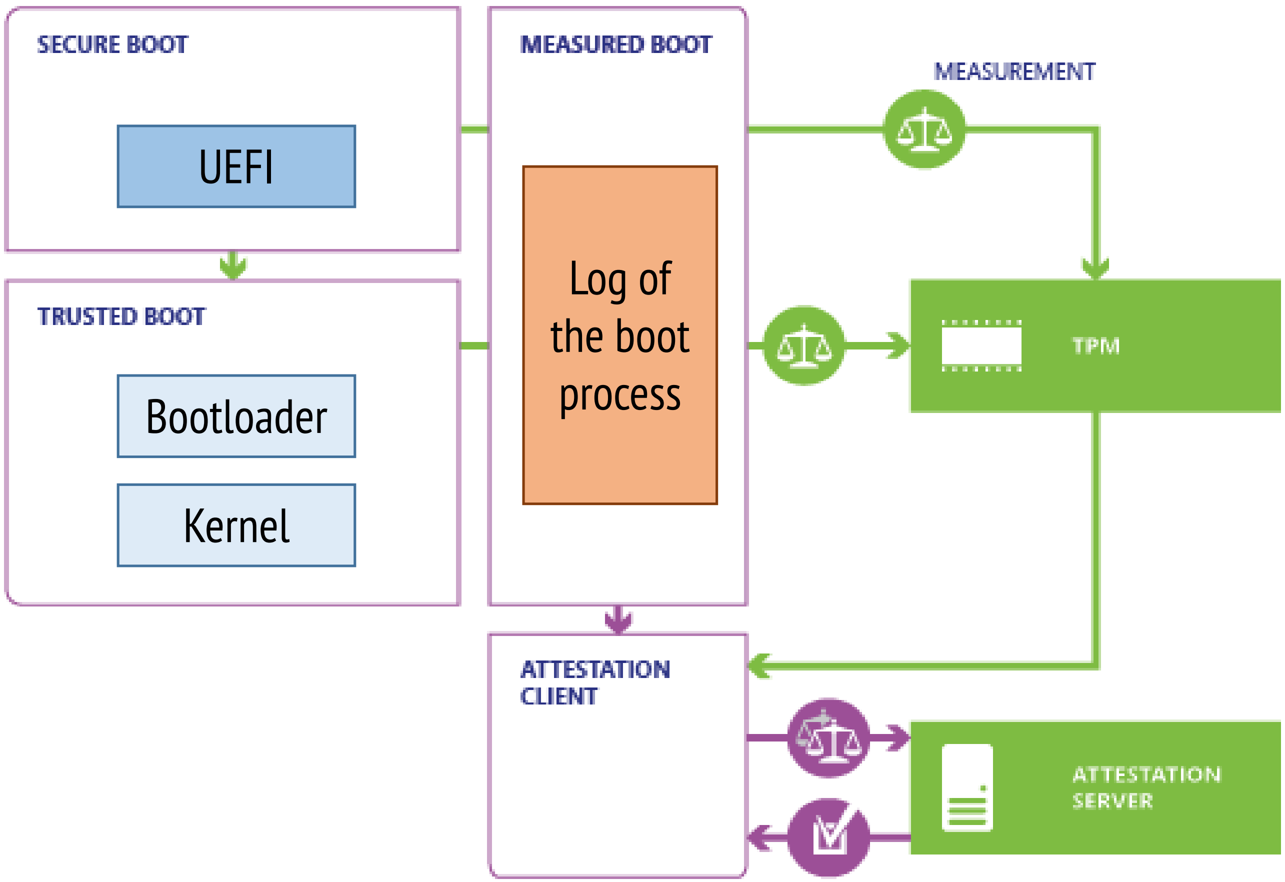
# Securing the boot process with a TPM?



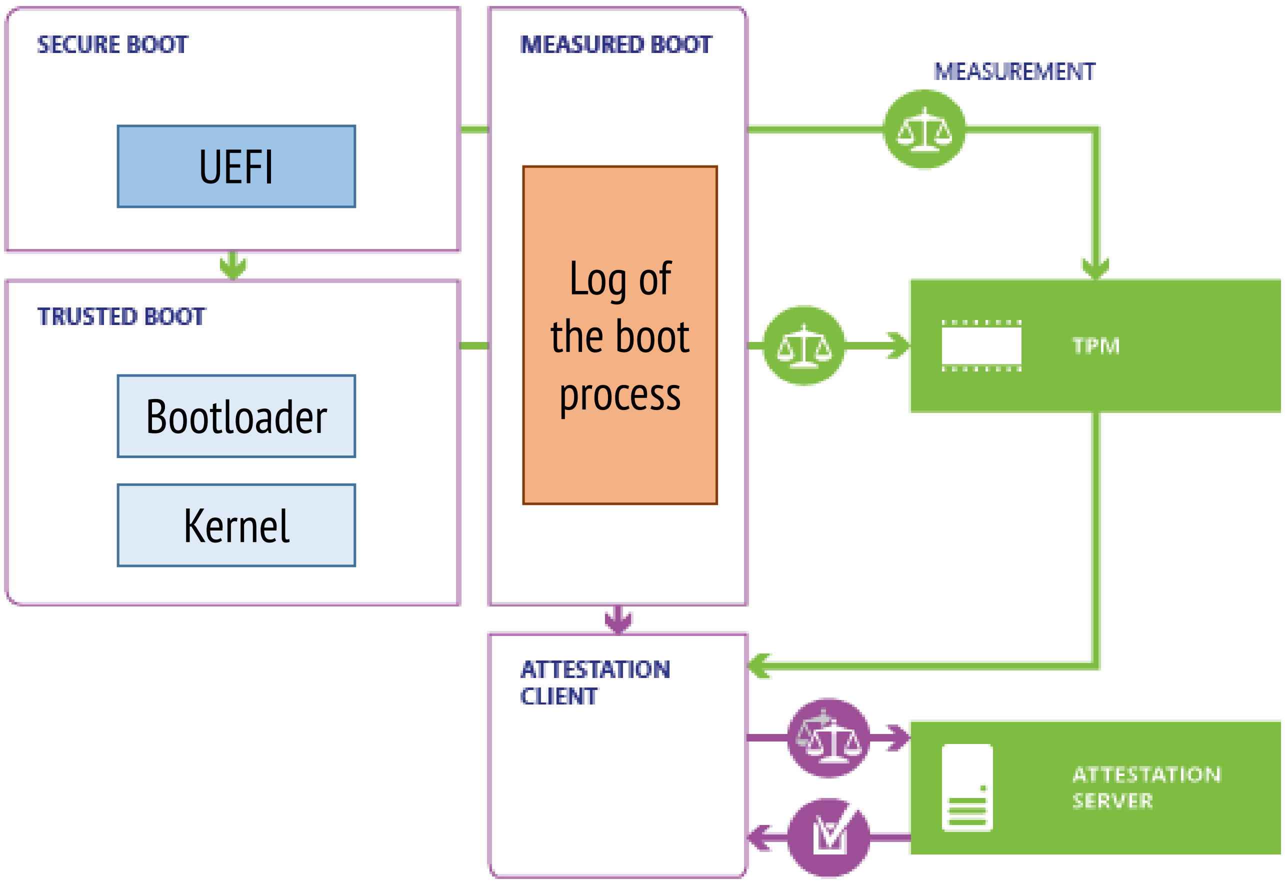
# Securing the boot process with a TPM?



# Securing the boot process with a TPM?



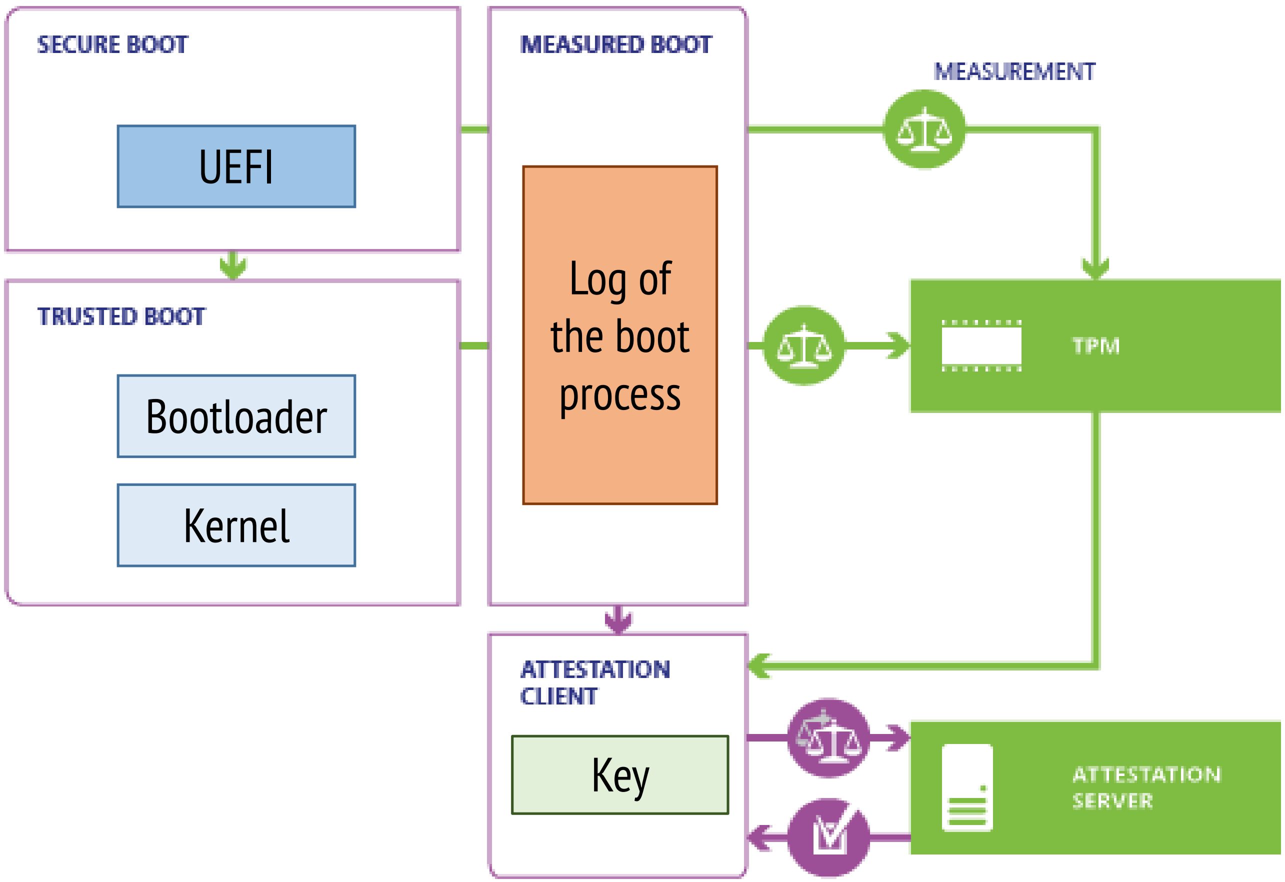
# Securing the boot process with a TPM?




Hashes for:

1. Firmware
2. Bootloader
3. Drivers
4. …

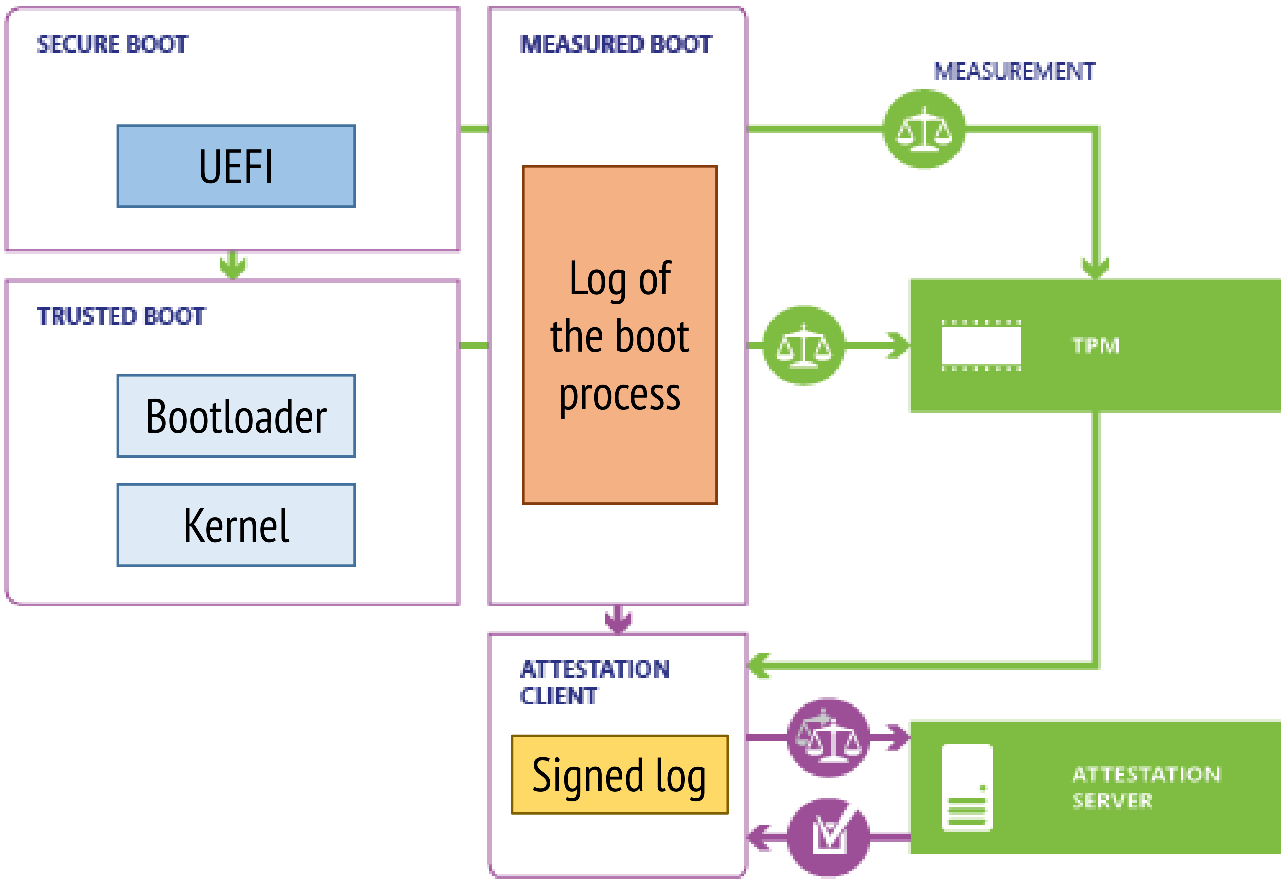
# Securing the boot process with a TPM?




Hashes for:

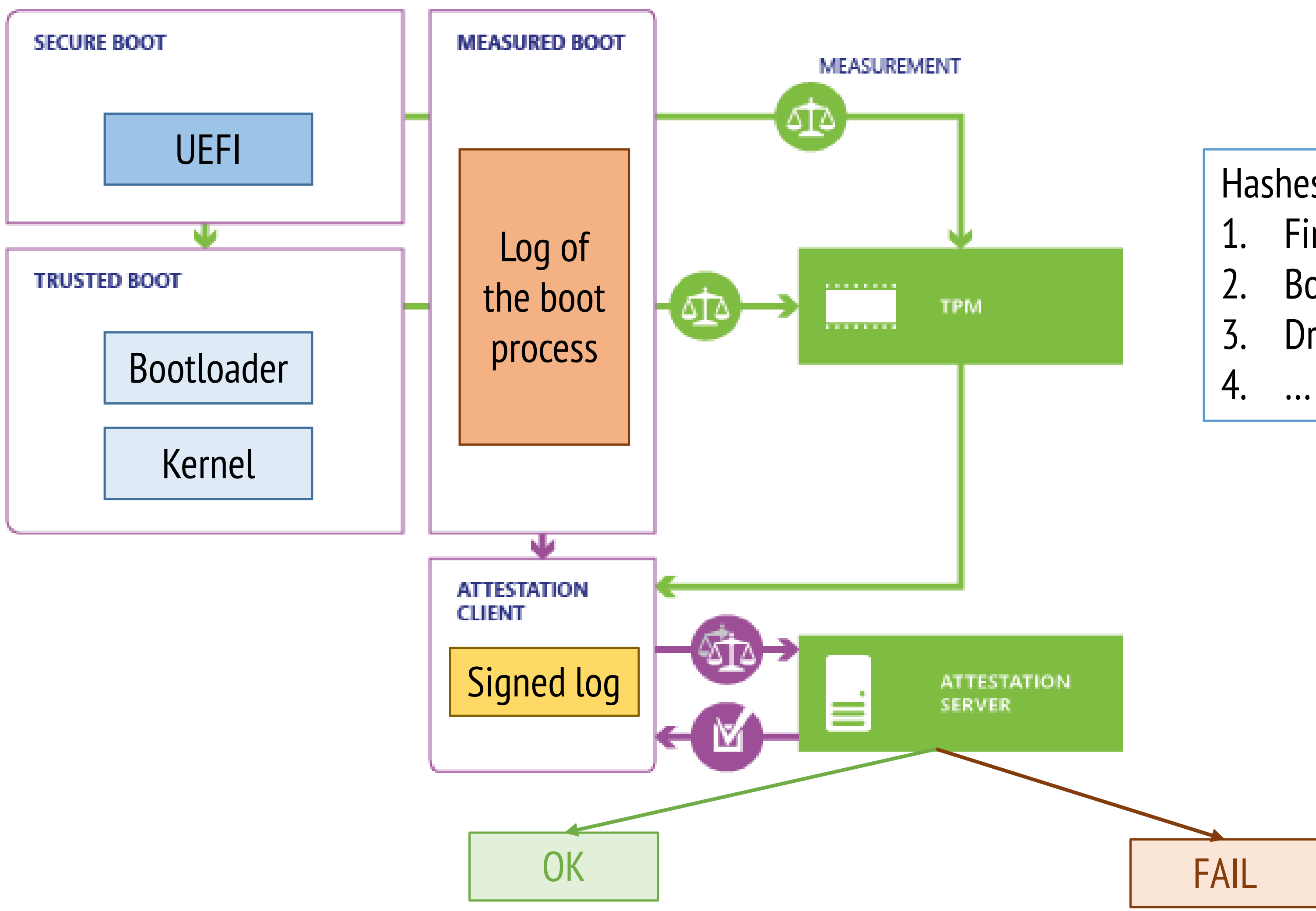
1. Firmware
2. Bootloader
3. Drivers
4. ...

# Securing the boot process with a TPM?




Hashes for:
1. Firmware
2. Bootloader
3. Drivers
4. …

# Securing the boot process with a TPM?

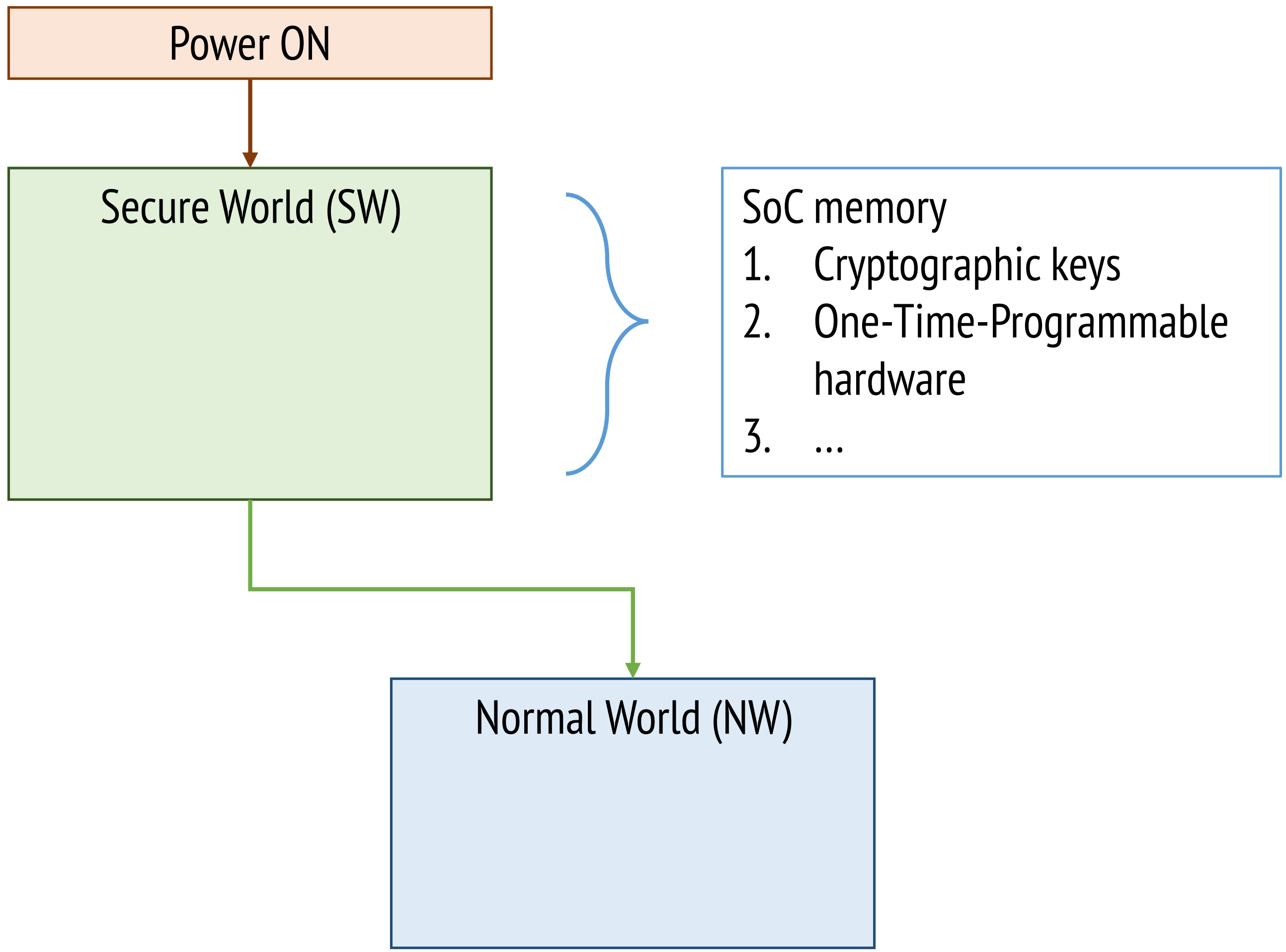



Hashes for:
1. Firmware
2. Bootloader
3. Drivers
4. …

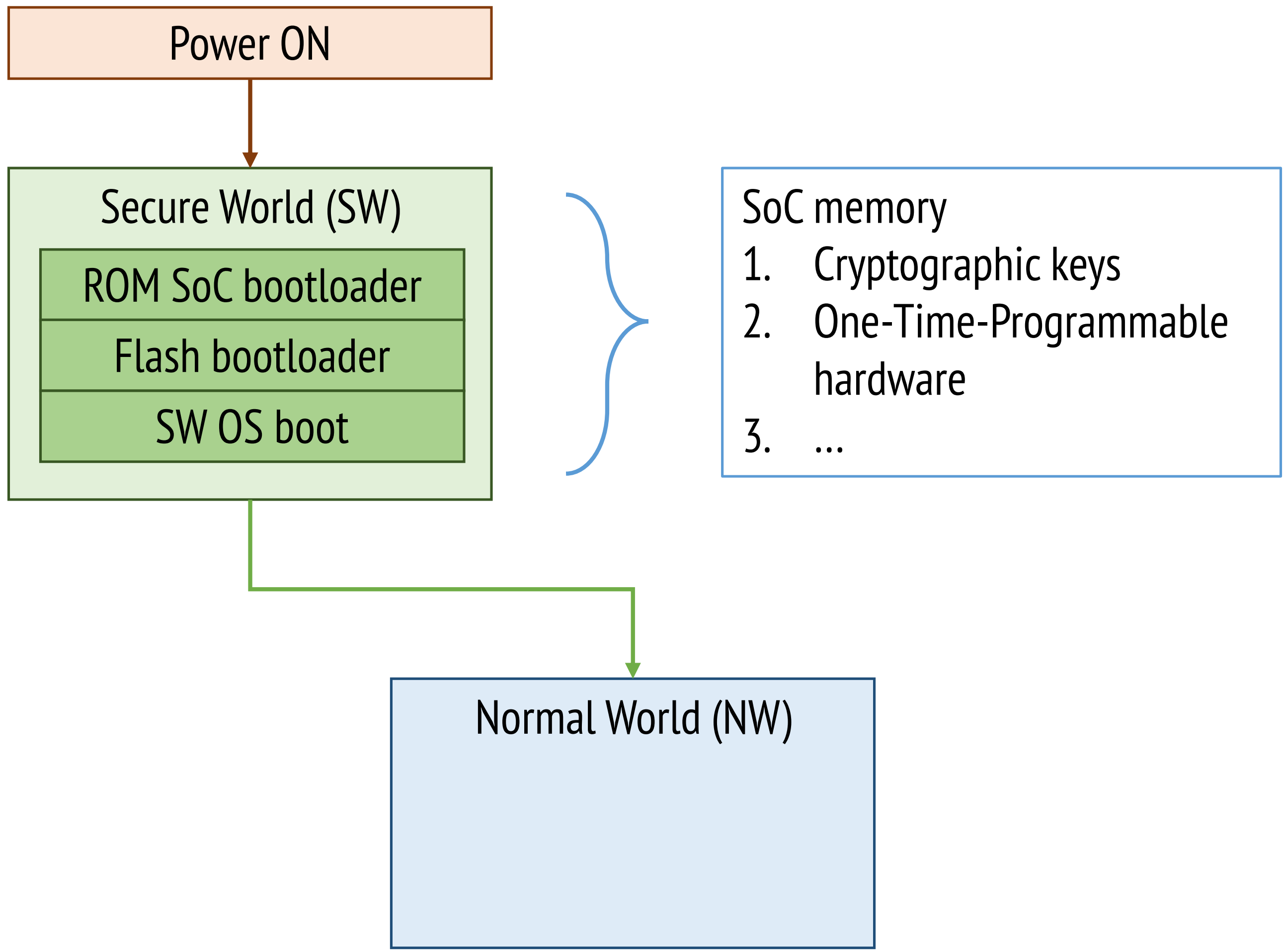
# Main principles applied

- Authenticate requests: the TPM authenticates UEFI application and the bootloader
- Fail secure: if things go wrong stop
- Audit and monitor: keep logs of the boot process and verify them against a known and trusted logs
- …

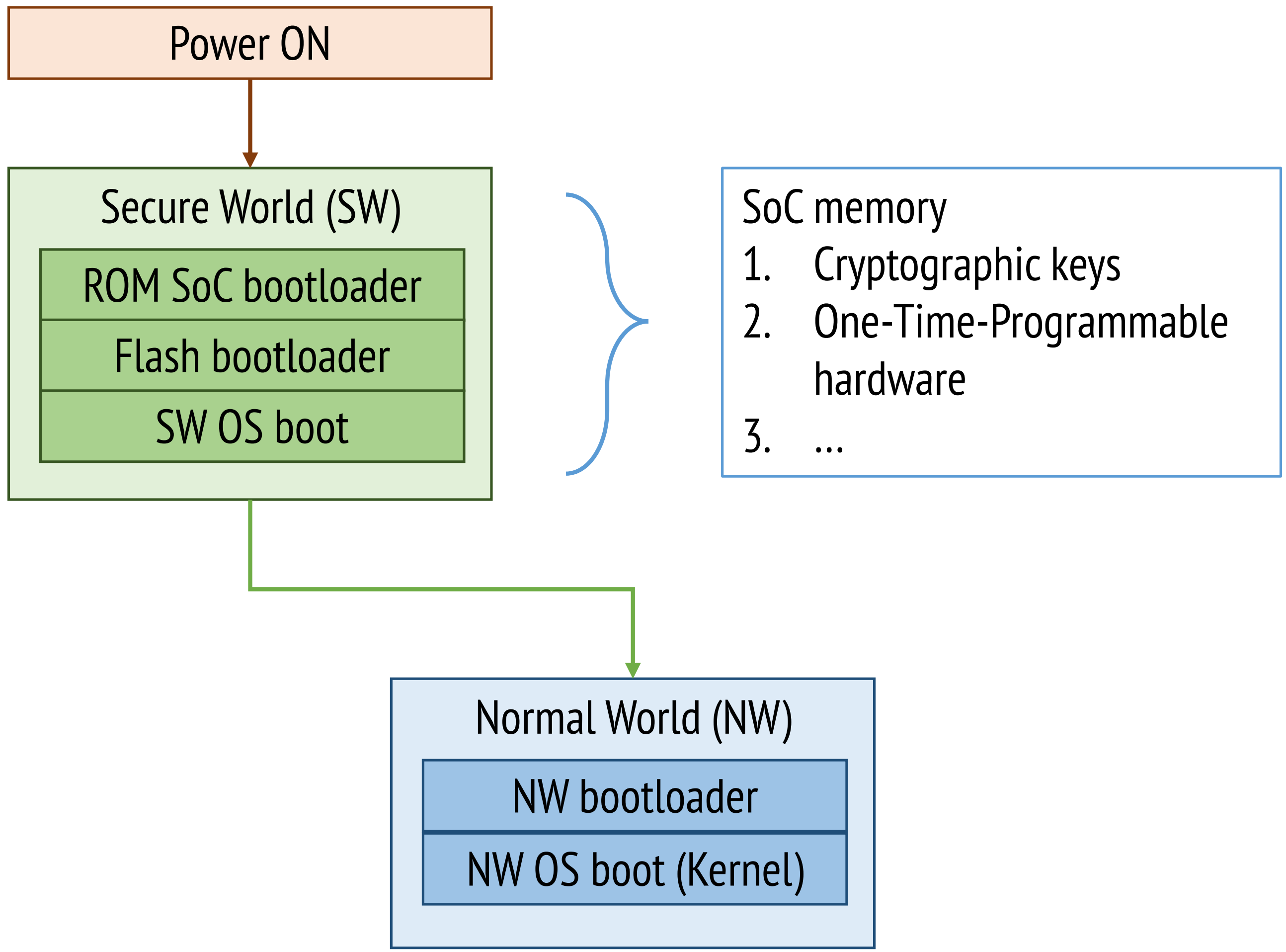
# Platform integrity with ARM TrustZone



# Platform integrity with ARM TrustZone



# Platform integrity with ARM TrustZone



# Main principles applied

- Economise mechanism: keep the system simple i.e. use a secure OS with controlled I/O to load the main kernel
- Fail secure: if things go wrong stop
- ...

# Secure Enclave Processor (SEP)

- Enable sensitive data to be stored securely
- Performs secure services for the rest of the SOC
- Runs its own operating system (SEPOS) which includes: kernel, drivers, services, and applications
- Supports multiple services: TouchID, ApplePay…

# Secure Enclave Processor (SEP)

Hardware functionality

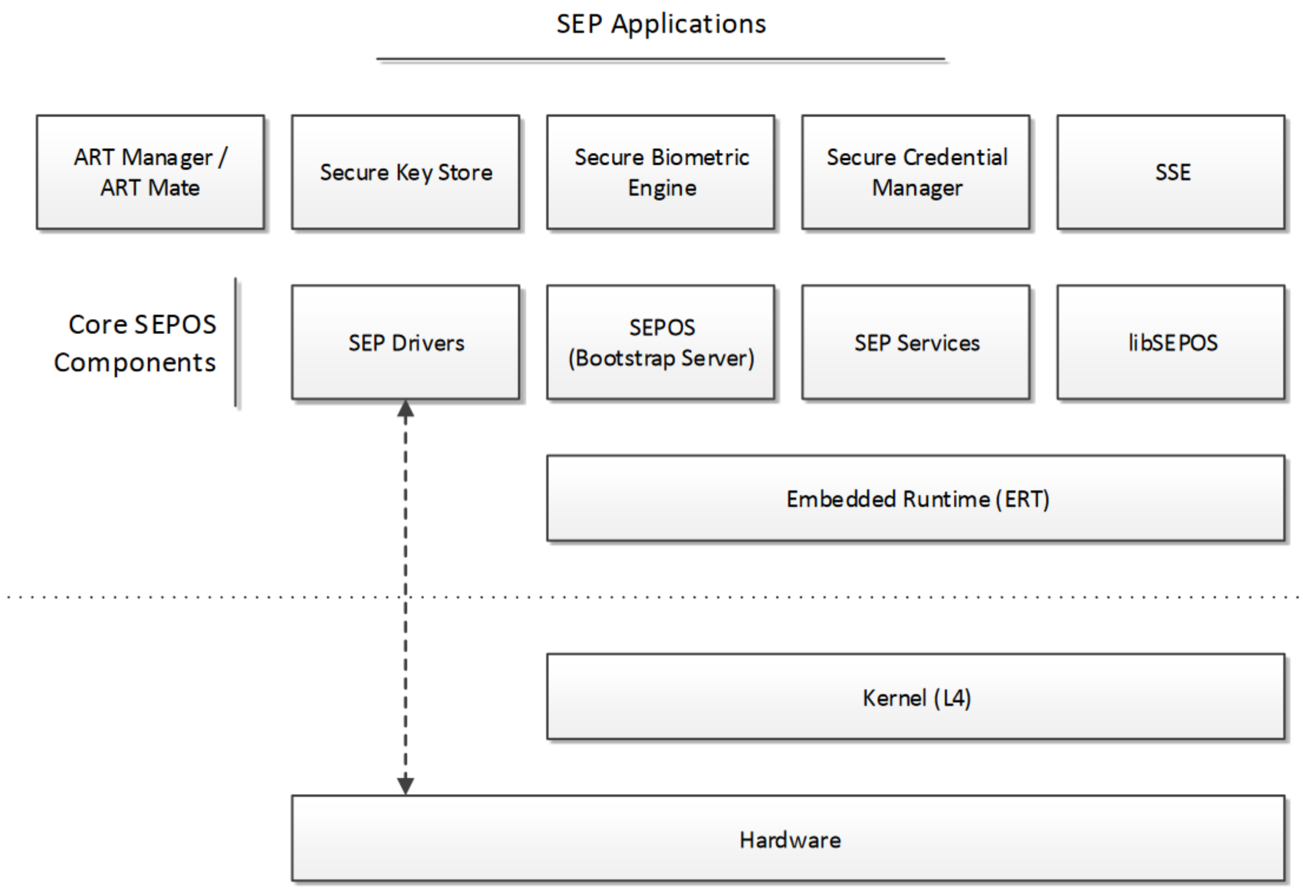
- Crypto engine
- Random Number Generator
- Fuses
- GID/UID
- Dedicated scratch RAM
- Hardware “filter” to prevent application processor (AP) to SEP memory access

# Secure Enclave Processor

Shared functionality with application processor

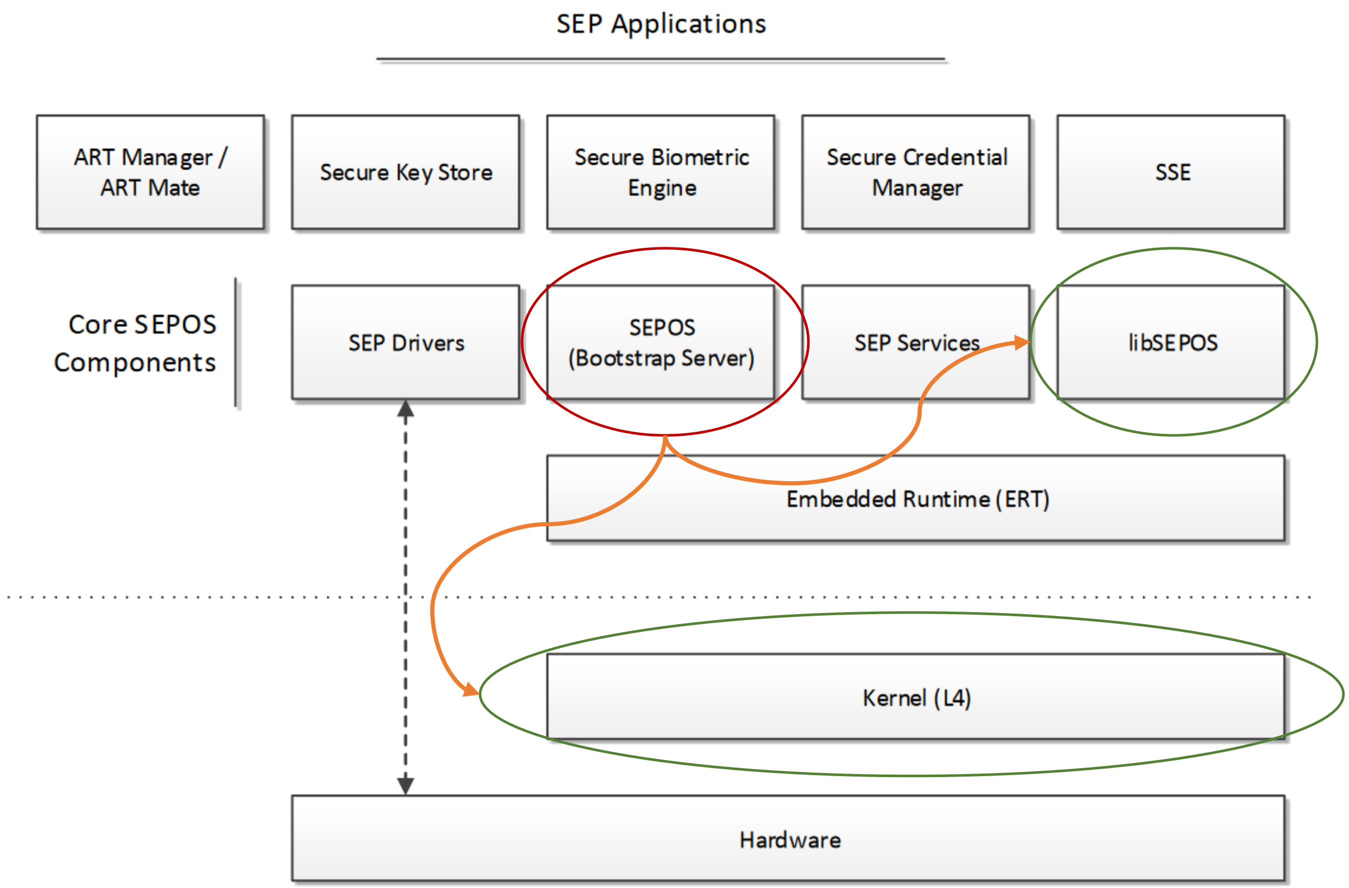
- Clock
- RAM
- Power manager
- ...

# SEP Architecture




https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

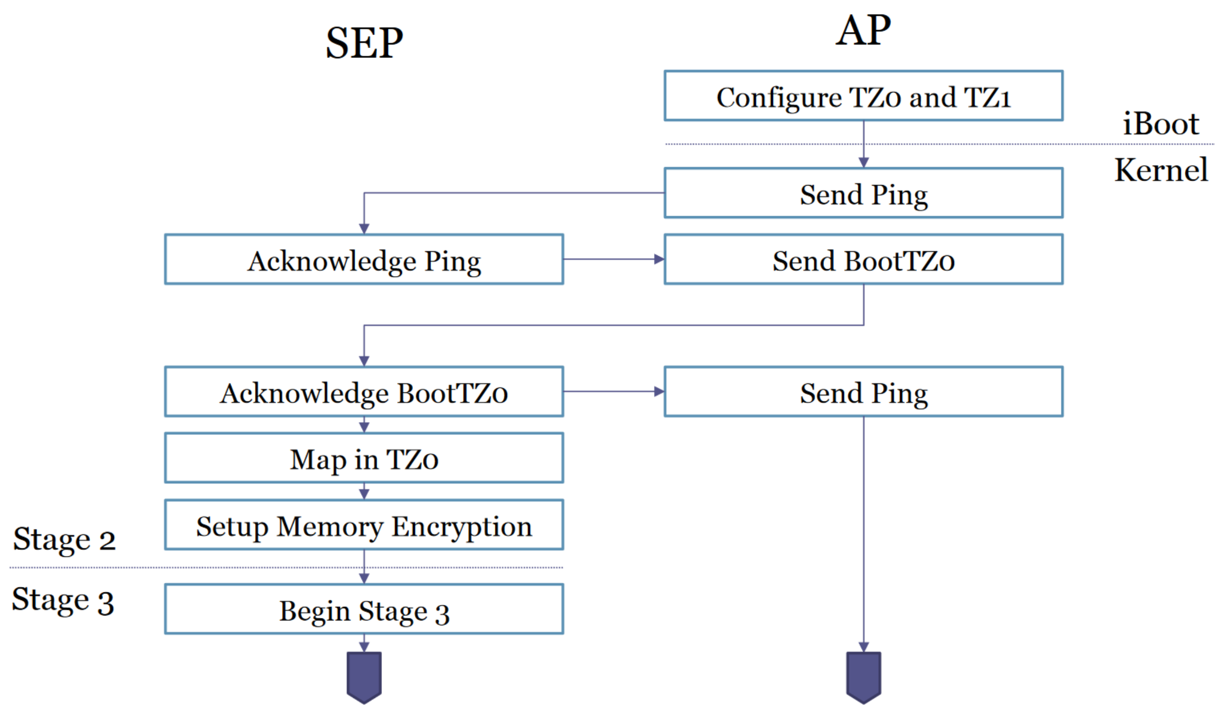
# SEP Architecture




https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

# Secure Enclave Processor (SEP)

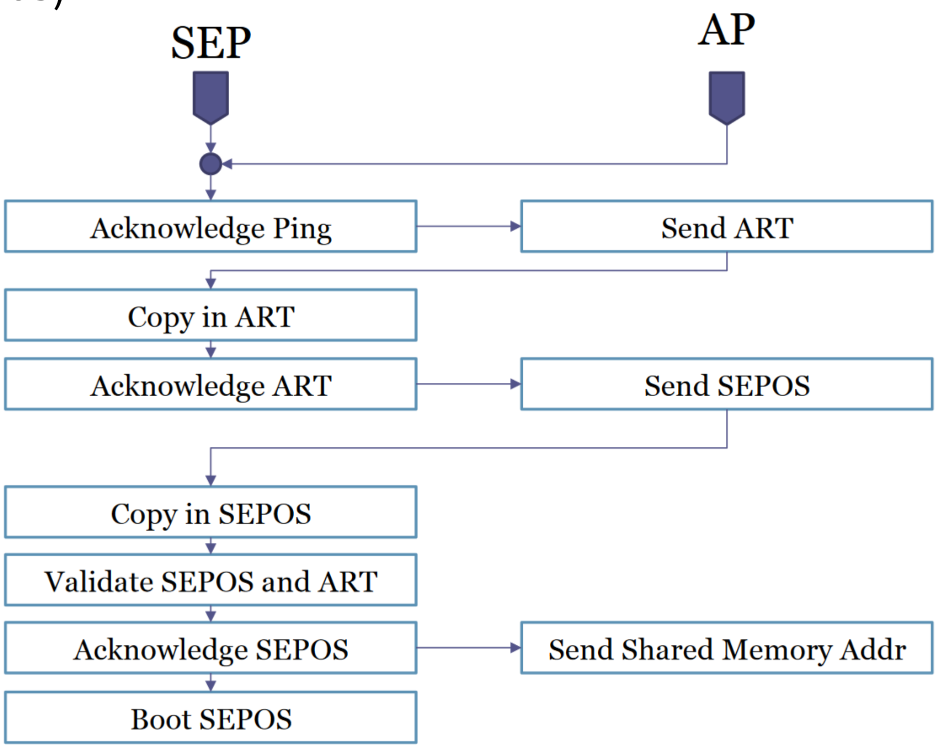
1. Configure Trust Zones 0/1
   (TZ0-SEP, TZ1-AP)
2. Check for SEP
3. Configure memory protection




https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

# Secure Enclave Processor (SEP)

1. Send anti-replay token (ART)
2. Verify ART
3. Verify the SEP operating system (SEPOS)
   i.e. 4096 bytes
4. Establish shared memory location

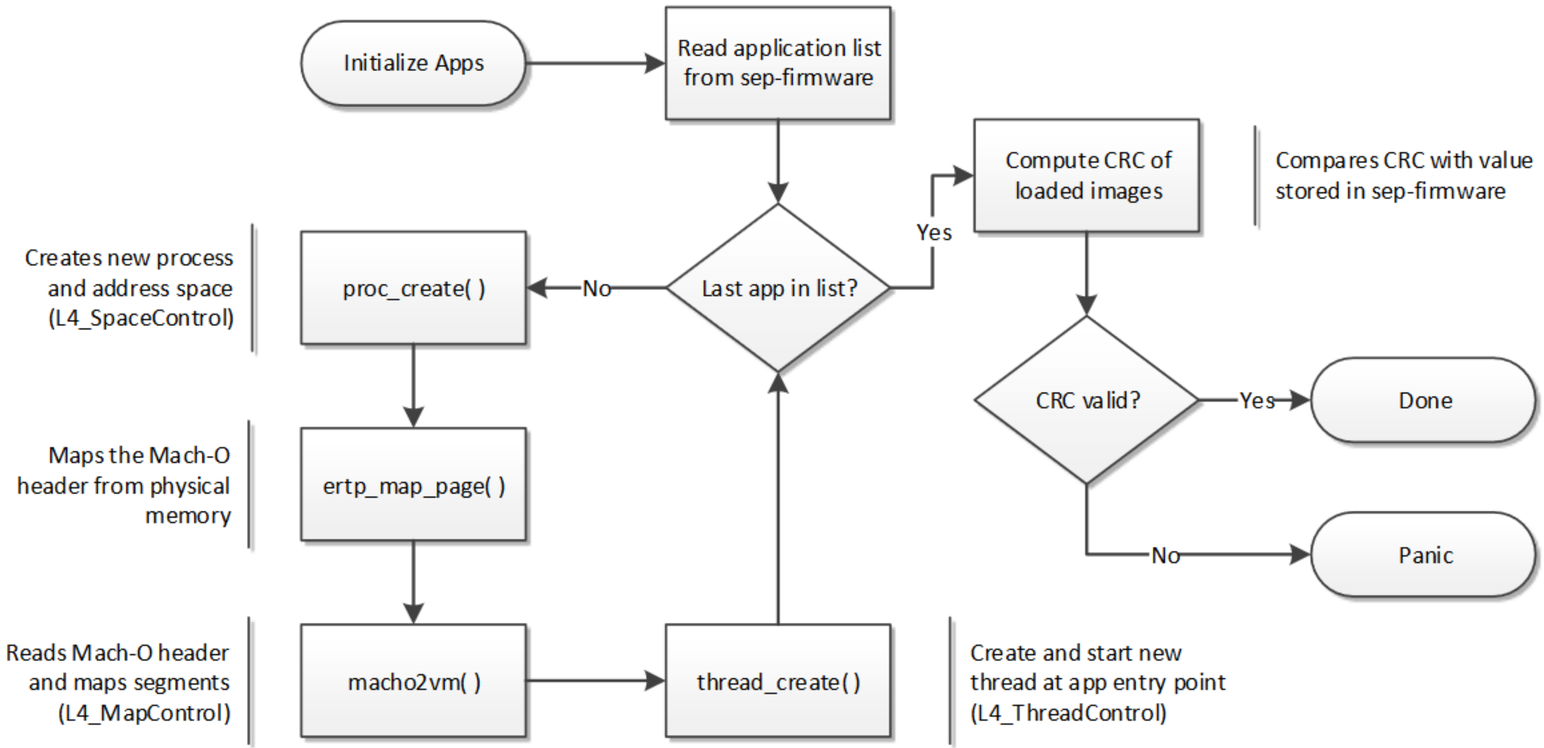



https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

# SEP communication

- Secure Mailbox allows the AP to communicate with the SEP
- Supported through the SEP Manager API
- Implemented using the SEP device I/O registers

# SEP interaction

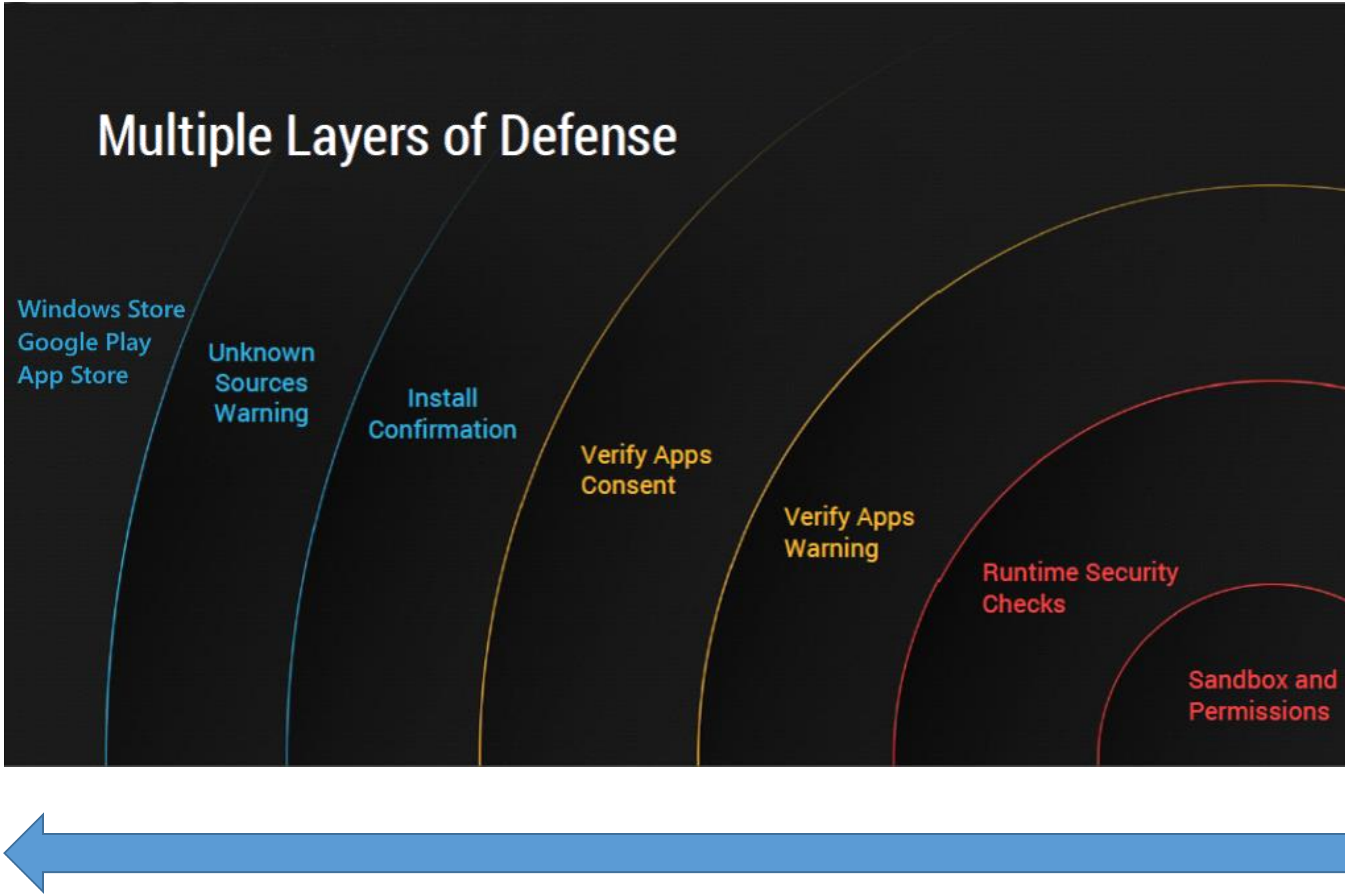



https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

# Main principles applied

- Authenticate requests: messages to the SEP are authenticated, apps are authenticated before use
- Fail secure: if anything goes wrong stop execution
- Segregation of duties: all cryptographic operations and accesses to secure hardware are intermediated by the SEP
- Secure the weakest link: protect access to memory

# Application security


Multiple Layers of Defense
Windows Store
Google Play
App Store
Unknown Sources Warning
Install Confirmation
Verify Apps Consent
Verify Apps Warning
Runtime Security Checks
Sandbox and Permissions

# Main “techniques”

- Code signing
- Runtime security
    a) Mandatory access control (MAC)
    b) Sandboxing
    c) Memory protection (e.g. address space layout randomisation (ASLR), ARM Execute Never (XN))

# Code signing

- Ensure applications have an approved source and haven't been tampered with
- Executable code is signed with store specific certificates
- Prevent applications from loading unsigned code resources and self-modifying code
- Access hardware with OS APIs

# Mandatory access control (MAC)

- Support over all OS
  - Selinux in Linux and Android
  - Mandatory Integrity Control in Windows Vista/7/8/10
  - TrustedBSD variant in Apple IOS and OSX

- MAC is usually enforced over all processes, even processes running with root/superuser privileges.

- MAC usually defaults to denial: anything that is not explicitly allowed is denied.

# Sandboxing

- Restrict applications from using data and resources from other apps.
- Each app has its own random “home directory”
- Run applications as non-privileged user
- Mount “important” partitions as read-only

# Memory protection

- Address space layout randomisation (ASLR)
  - protects memory from corruption
  - protects against attacks that target the stack and/or memory addresses
- Hardware support like ARM’s Execute never
  - Supports “permissions” for memory pages
  - Marks memory pages as non-executable

# Conclusions

- Multiple surfaces of attack
- Root of trust

- Defence in depth
- Access control
- Audit and monitor
- Make security usable
- Authenticate requests
- Control access
- ...