Aalto University

Platform Security Lecture 5: Measurement and Attestation

You will be learning:

Booting a PC

- x86 boot process
- Secure boot
- Authenticated boot

Trusted platform module

- Measuring the system
- Authorisation

Remote attestation

- Types of attestation
- Different implementations

System State

Problem: How do we know whether a system is in the right state?

Two parts:

- How to get the system into the right place to begin with?
- How to validate the current state of a system?

Solution: Secure/measured boot, remote attestation

Booting the system

The boot process

Booting a computer is a multi-stage process

- Firmware
- Bootloader
- OS kernel
- System software
- User software

Starting the x86 boot process

When an x86 system is powered up, it goes through several steps

1. **CPU jumps to address 0xFFFFFFF0**
   - Code at this address is known as the “reset vector”
   - Stored in (slow) non-volatile memory

2. **Reset vector jumps to initialisation code**
   - Configures CPU and memory
   - Sets up the stack
   - Copies code to [fast] main memory and continues execution from there

3. **More initialisation code run from main memory**
   - Keyboard, mouse, display, storage
   - Loads bootloader from storage, hands over control

This is firmware: code embedded in the motherboard that performs hardware-specific functionality.

a.k.a. the Basic Input/Output Subsystem (BIOS)

https://www.intel.com/content/www/us/en/intelligent-systems/intel-boot-loader-development-kit/minimal-intel-architecture-boot-loader-paper.html

Traditional bootloader

BIOS loads and runs bootloader from Master Boot Record (MBR)

- First 446 bytes of the boot device contain bootloader's initial code

MBR loads and runs rest of bootloader from disk

- BIOS provides drivers for bootloader to access disk

```
start:
  cli                           ; We do not want to be interrupted
  xor ax, ax                    ; 0 AX
  mov ds, ax                    ; Set Data Segment to 0
  mov es, ax                    ; Set Extra Segment to 0
  mov ss, ax                    ; Set Stack Segment to 0
  mov sp, ax                    ; Set Stack Pointer to 0
  .CopyLower:
    mov cx, 0x0100              ; 256 WORDs in MBR
    mov si, 0x7C00              ; Current MBR Address
    mov di, 0x0600              ; New MBR Address
    rep movsw                   ; Copy MBR
  jmp 0:LowStart                ; Jump to new Address

LowStart:
  sti                           ; Start interrupts
  mov BYTE [bootDrive], dl      ; Save BootDrive
  .CheckPartitions:             ; Check Partition Table For Bootable Partition
    mov bx, PT1                 ; Base = Partition Table Entry 1
    mov cx, 4                   ; There are 4 Partition Table Entries
```

```
.CKPTloop:
     mov al, BYTE [bx]          ; Get Boot indicator bit flag
     test al, 0x80              ; Check For Active Bit
     jnz .CKPTFound             ; We Found an Active Partition
     add bx, 0x10               ; Partition Table Entry is 16 Bytes
     dec cx                     ; Decrement Counter
     jnz .CKPTloop              ; Loop
jmp ERROR                    ; ERROR!
   .CKPTFound:
     mov WORD [PToff], bx       ; Save Offset
     add bx, 8                  ; Increment Base to LBA Address
 .ReadVBR:
   mov EBX, DWORD [bx]          ; Start LBA of Active Partition
   mov di, 0x7C00               ; We Are Loading VBR to 0x07C0:0x0000
   mov cx, 1                    ; Only one sector
   call ReadSectors             ; Read Sector

 .jumpToVBR:
   cmp WORD [0x7DFE], 0xAA55 ; Check Boot Signature
   jne ERROR                    ; Error if not Boot Signature
   mov si, WORD [PToff]         ; Set DS:SI to Partition Table Entry
   mov dl, BYTE [bootDrive]     ; Set DL to Drive Number
   jmp 0x7C00                   ; Jump To VBR
```

Example: https://wiki.osdev.org/MBR_(x86)

Universal Extensible Firmware Interface (UEFI)

Firmware specification to replacement the traditional BIOS

- Consistent interfaces for much more functionality (filesystem access, USB, etc.)

Traditional bootloaders replaced with UEFI Applications

- No more squeezing code into the MBR!

Supports secure boot (more next week)

From kernel code to user code

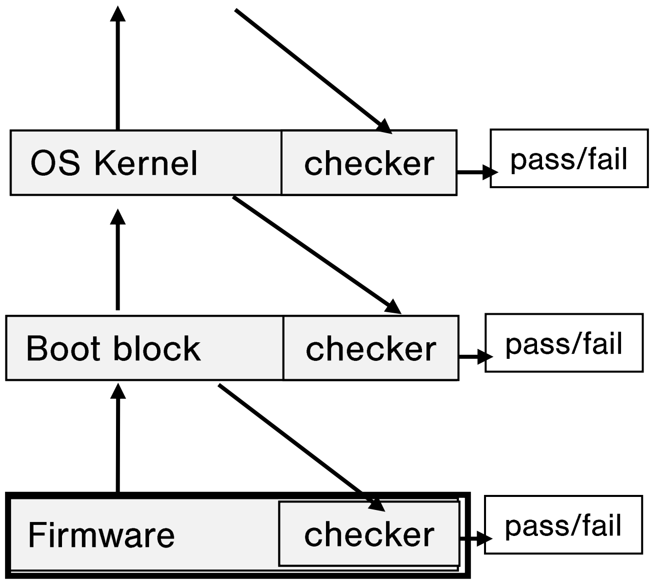
Unix-like (old way: sysvinit)

- Kernel runs `/sbin/init`
- `/sbin/init` reads /etc/inittab which points to a script to run

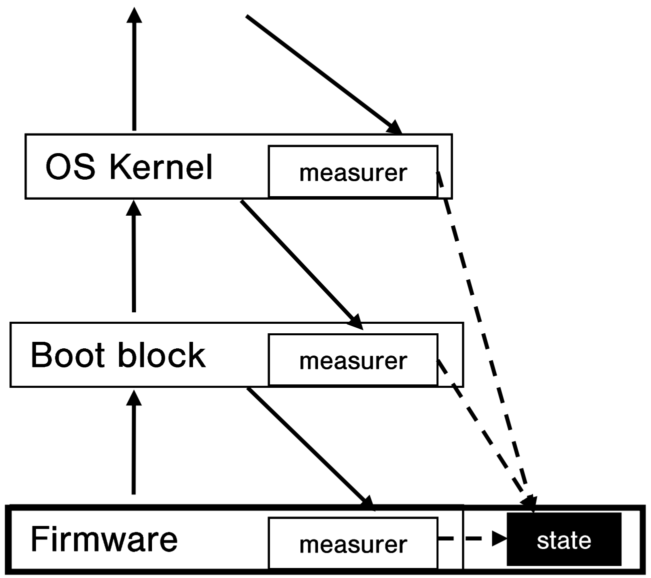
Unix-like (new way: systemd)

- Kernel runs systemd
- Large piece of software that manages system software directly

Secure boot vs authenticated boot



Secure boot



Authenticated boot

Chain of trust for secure & authenticated boot

Both approaches require a chain of trust

1. Root of trust for measurement (RTM) must be trusted to measure the firmware

- The RTM is the first code run on the main processor

2. The firmware must be trusted to measure the bootloader

- We can trust the firmware because it was measured

3. The bootloader must be trusted to measure the OS

- *On Linux we normally stop here*

4. The OS must be trusted to measure applications

Authenticated Boot using Trusted Platform Modules

What is a TPM?

Collects state information about a system

- Separate from system on which it reports

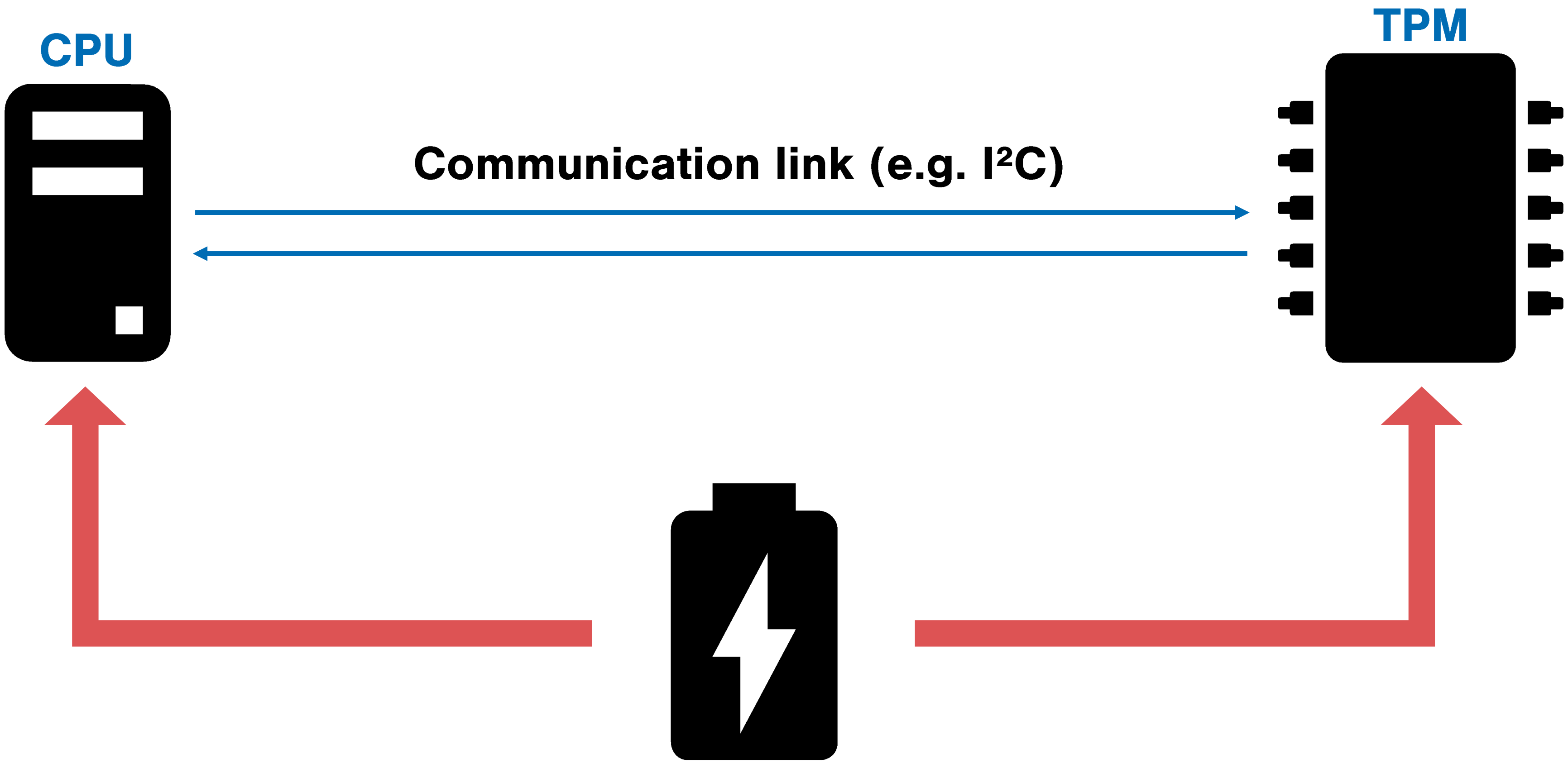
For remote parties

- Well-defined remote attestation
- Authorisation for functions/objects in TPM

Locally

- Generation/use of TPM-resident keys
- Sealing: Securing data for **non-volatile storage** (w/ binding)
- Engine for cryptographic operations

TPM model



TPM is powered up/reset at the same time as CPU

Critical that TPM cannot be independently reset

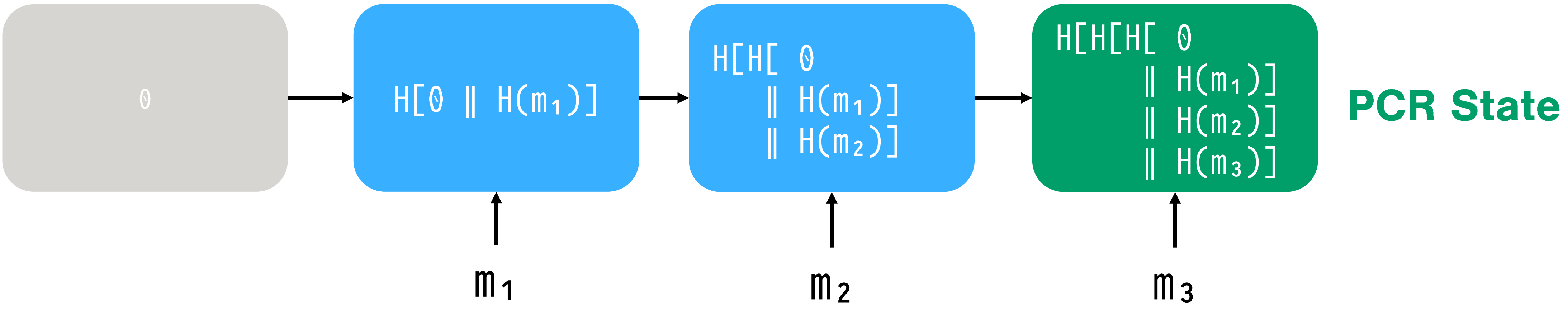
Platform Configuration Registers

Integrity-protected registers

- In TPM volatile memory
- Values represent current system configuration

PCRs store aggregated measurement of platform state

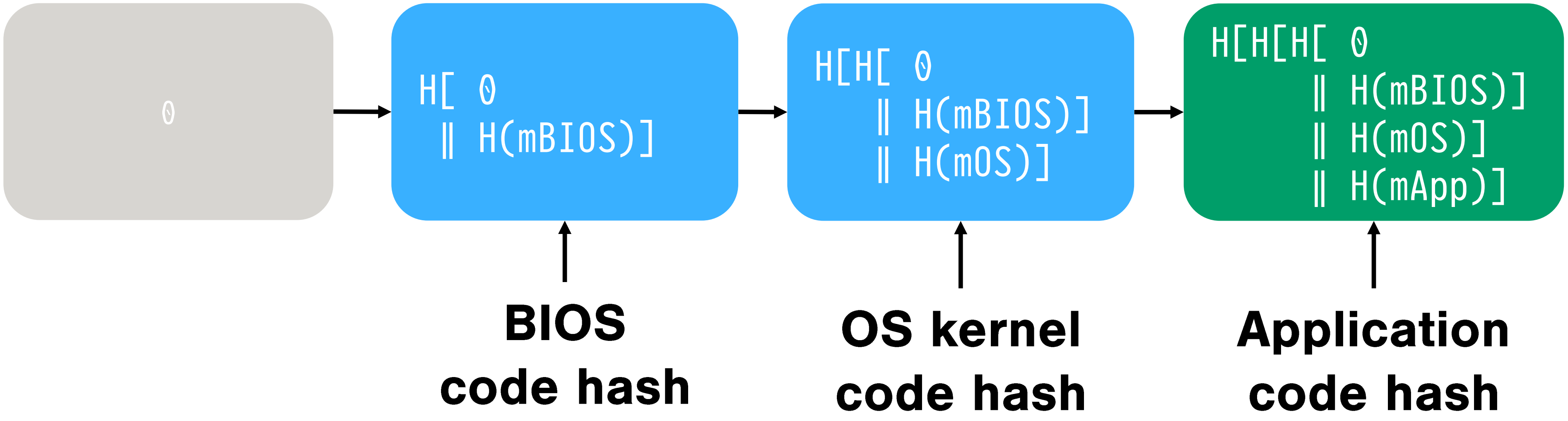
- Goal: PCR value represents whole sequence of measurements
- Modifications by *extension*: $\mathtt{PCR} \leftarrow \mathtt{H}(\mathtt{PCR} \parallel \mathit{digest})$



Trusted Platform Module Library Part 1: Architecture

Measurements can include e.g. code hashes

- You will see examples of different kinds of measurement later



[Trusted Platform Module Library Part 1: Architecture]

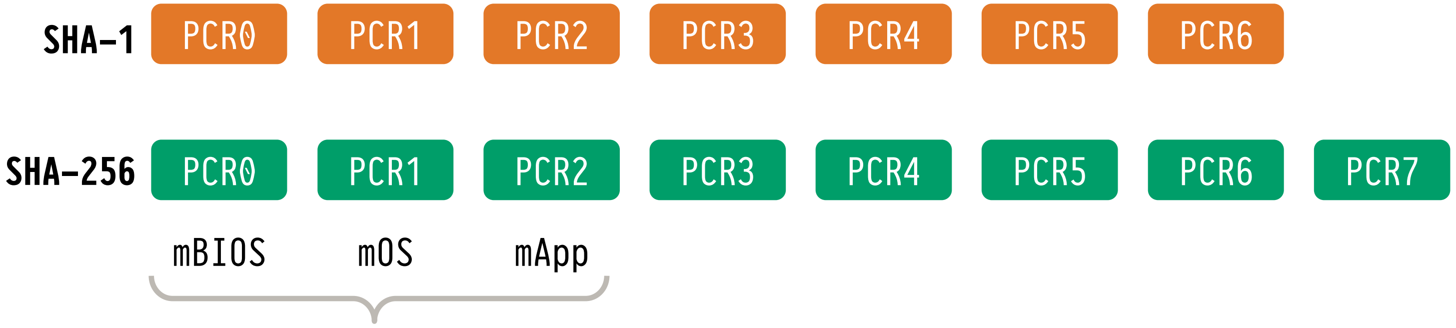
Platform Configuration Registers

Problem: Changes to any measurement change the aggregate measurement

- Software updates lead to huge numbers of valid measurements

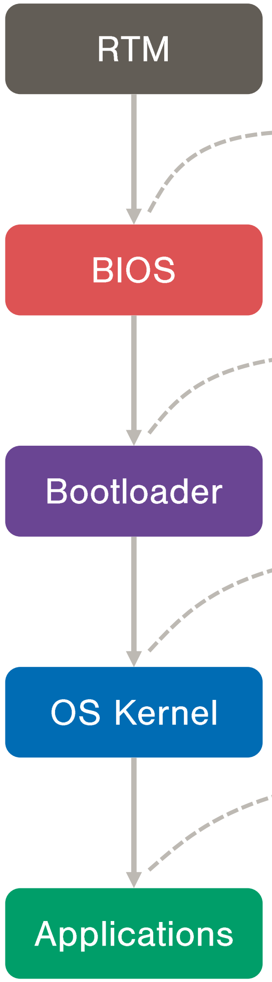
TPM contains multiple PCRs that can be used for different purposes

- Separate banks of PCRs are used for each hash algorithm



NB: Assignments are only illustrative

Back to measured boot



$$H[0 \parallel H(mBIOS)]$$

$$H[H[0 \parallel H(mBIOS)] \parallel H(mBootLoader)]$$

$$H[H[H[0 \parallel H(mBIOS)] \parallel H(mBootLoader)] \parallel H(mKernel)]$$

$$H[H[H[H[0 \parallel H(mBIOS)] \parallel H(mBootLoader)] \parallel H(mKernel)] \parallel H(mApp)]$$

Back to measured boot

	PCRa	PCRb	PCRc	PCRd
RTM				
BIOS	H[0 ‖ H(mBIOS)]	0	0	0
Bootloader	H[0 ‖ H(mBIOS)]	H[0 ‖ H(mBootLoader)]	0	0
OS Kernel	H[0 ‖ H(mBIOS)]	H[0 ‖ H(mBootLoader)]	H[0 ‖ H(mKernel)]	0
Applications	H[0 ‖ H(mBIOS)]	H[0 ‖ H(mBootLoader)]	H[0 ‖ H(mKernel)]	H[0 ‖ H(mApp)]

Late launch

Observation: BIOS code not used after jump to OS

Late Launch allows the CPU to jump to a dynamic root of trust

- CPU measures block of code in memory, then runs it
- Some PCRs are reset when this happens
- Requires a firmware TPM, implemented inside the CPU itself

Result: Dynamic PCRs describe system state using just OS and application software

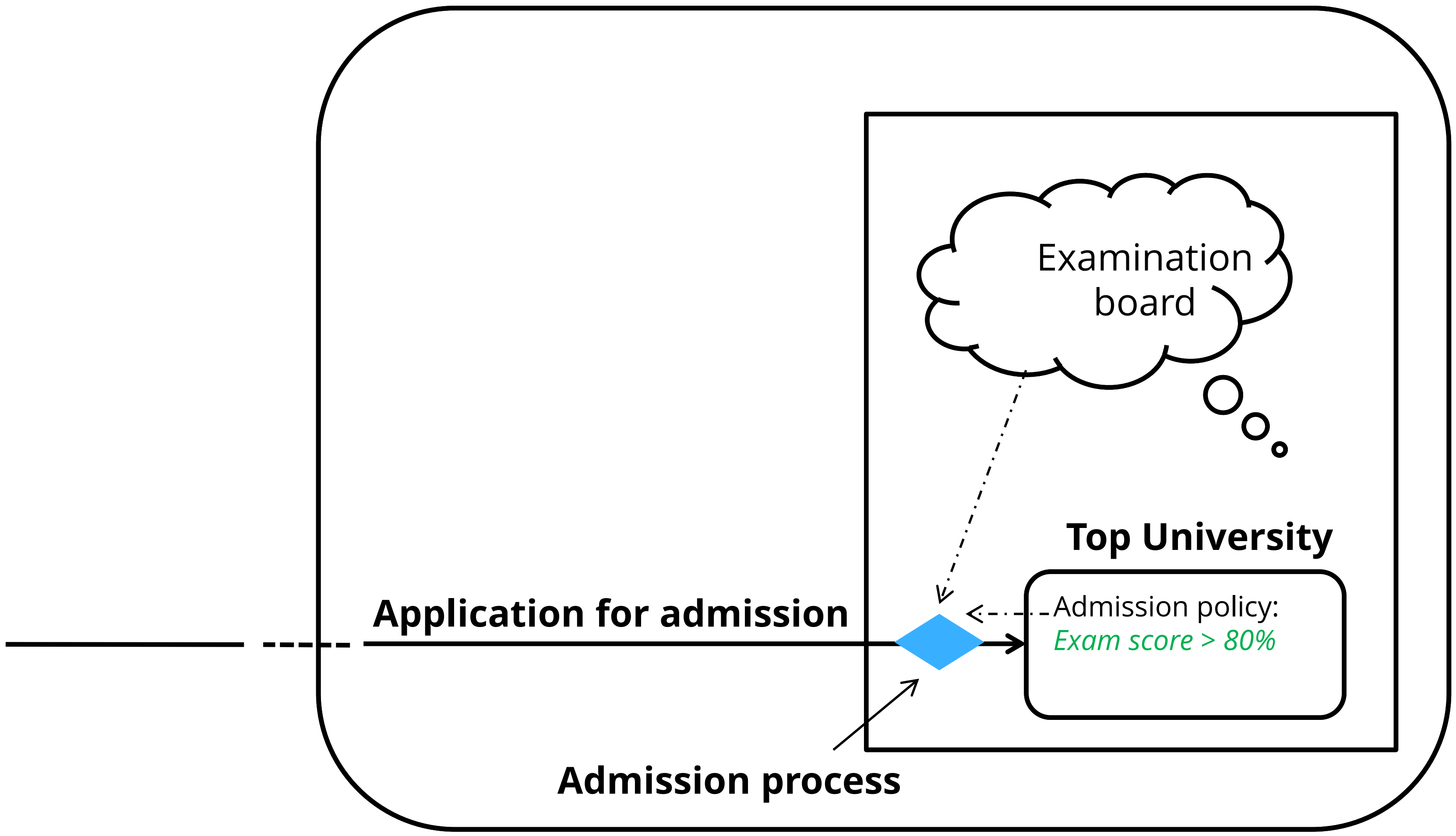
Objects in the TPM can have an access control policy attached

- Object is only usable if TPM is in the correct state
- Policy can include PCR values, counter values (to prevent password guessing)

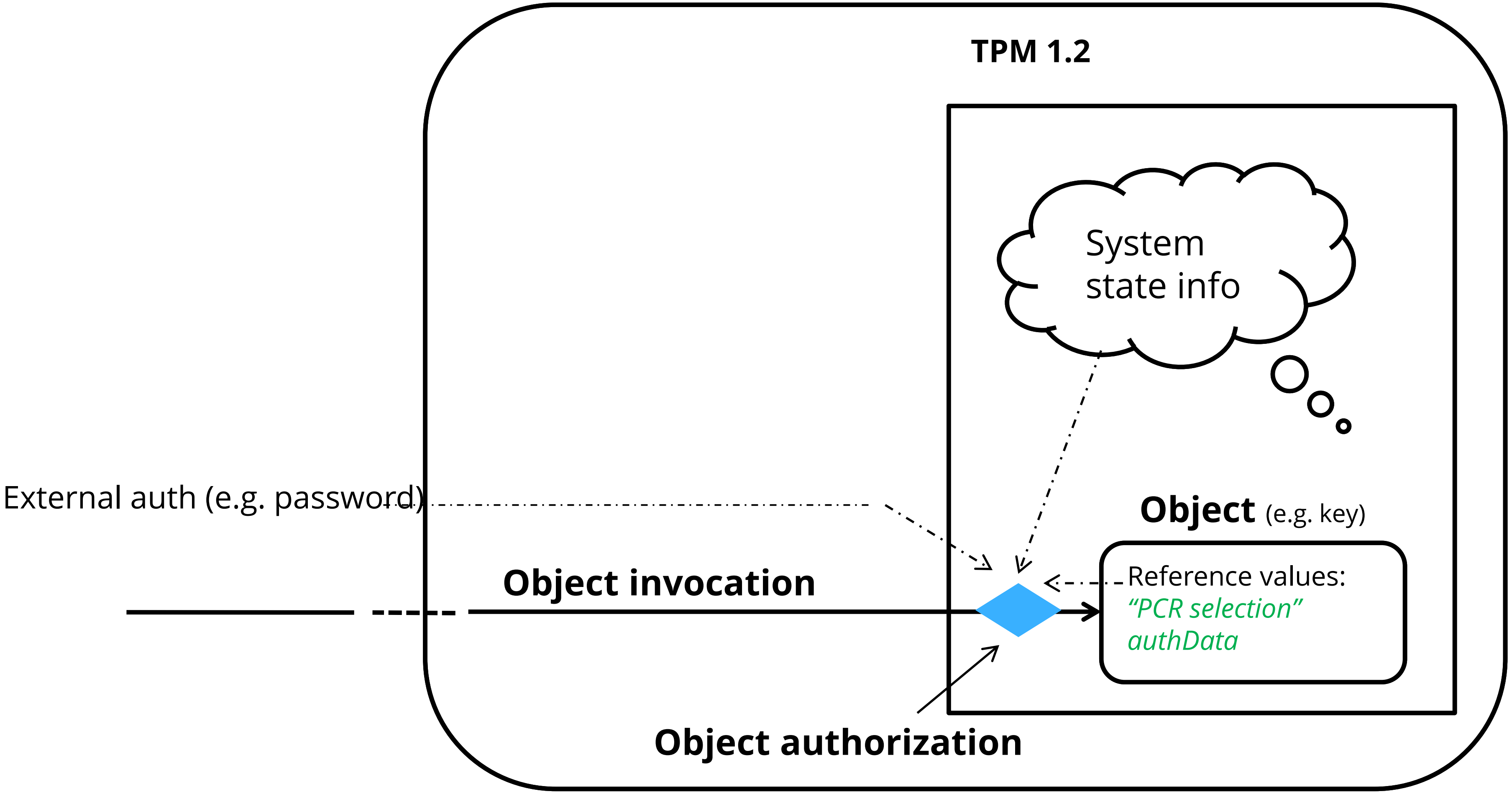
```
PCR = H[H[H[ 0
          ‖ H(mBIOS)]
          ‖ H(mOS)]
          ‖ H(mApp1)]
```

```
PCR = H[H[H[ 0
          ‖ H(mBIOS)]
          ‖ H(mOS)]
          ‖ H(mApp2)]
```

Authorisation policy analogy



TPM 1.2 authorisation policy



TPM sealing

Problem: Not enough storage in the TPM

- Only space to store a few keys
- Most data needs to be stored on a hard disk

Solution: Sealing

- TPM encrypts & authenticates data using a storage root key (SRK)
- Data is accompanied by authentication policy

Example:

1. Create RSA keypair pk/sk when PCR_X is Y
2. Bind private key: $Enc_{SRK}(sk, PCR_X{=}Y)$
3. TPM will “unseal” key **iff** PCR_X value is Y
    - Y is the “reference value”

TPM 2.0 Extended Authorisation

Specific PCR values aren't always flexible enough

- What if multiple configurations are acceptable?
- What about software updates?

TPM 2.0 supports more complex policies

- AND, OR, external authorisation

Uses a policy session that accumulates all authorisation information

- Performing a check extends the session's policyDigest
- Checks can be performed immediately, or later (deferred checks)
- Example of a deferred check: PolicyCommandCode limits the type of access to an object
    - e.g. a key can be used to encrypt, but never to decrypt

TPM 2.0 Extended Authorisation Example

```
Command                    PolicyDigest
TPM2_PolicyPCR(0, mBIOS)
                           H[0 || TPM_CC_PolicyPCR || 0 || H(mBIOS)]

TPM2_PolicyPCR(1,   mOS)
                           H[H[0 || TPM_CC_PolicyPCR || 0 || H(mBIOS)]
                                 || TPM_CC_PolicyPCR || 1 || H(mOS)]

TPM2_PolicyPCR(2,   mApp)

                           H[H[H[0 || TPM_CC_PolicyPCR || 0 || H(mBIOS)]
                                   || TPM_CC_PolicyPCR || 1 || H(mOS)]
                                   || TPM_CC_PolicyPCR || 2 || H(mApp)]
```

Policy disjunction

TPM2_PolicyOR: Authorize one of several options:

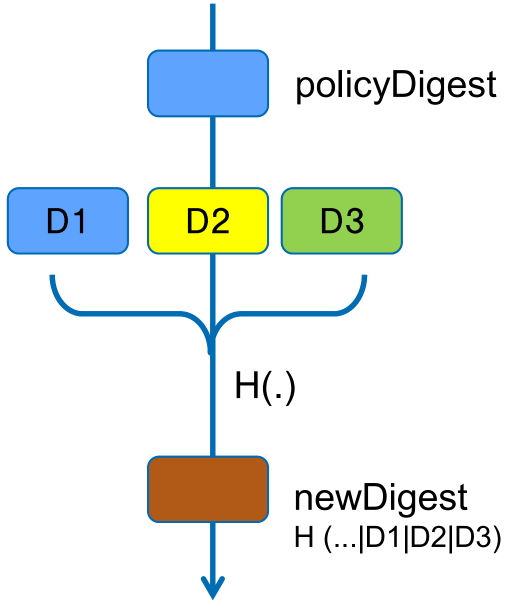
Input: *List* of digest values <D1, D2, D3, .. >

IF *policySession->policyDigest* in *List* **THEN**

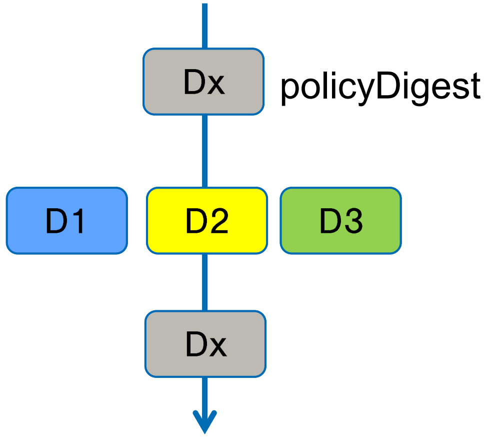
newDigest := H(0 || TPM2_CC_PolicyOR || $\boxed{\text{List)}}$

Reasoning: For a wrong digest Dx (not in <D1 D2 D3>) difficult to find *List2* = <Dx Dy, Dz, .. >

such that H(... |List) == H(... |List2)

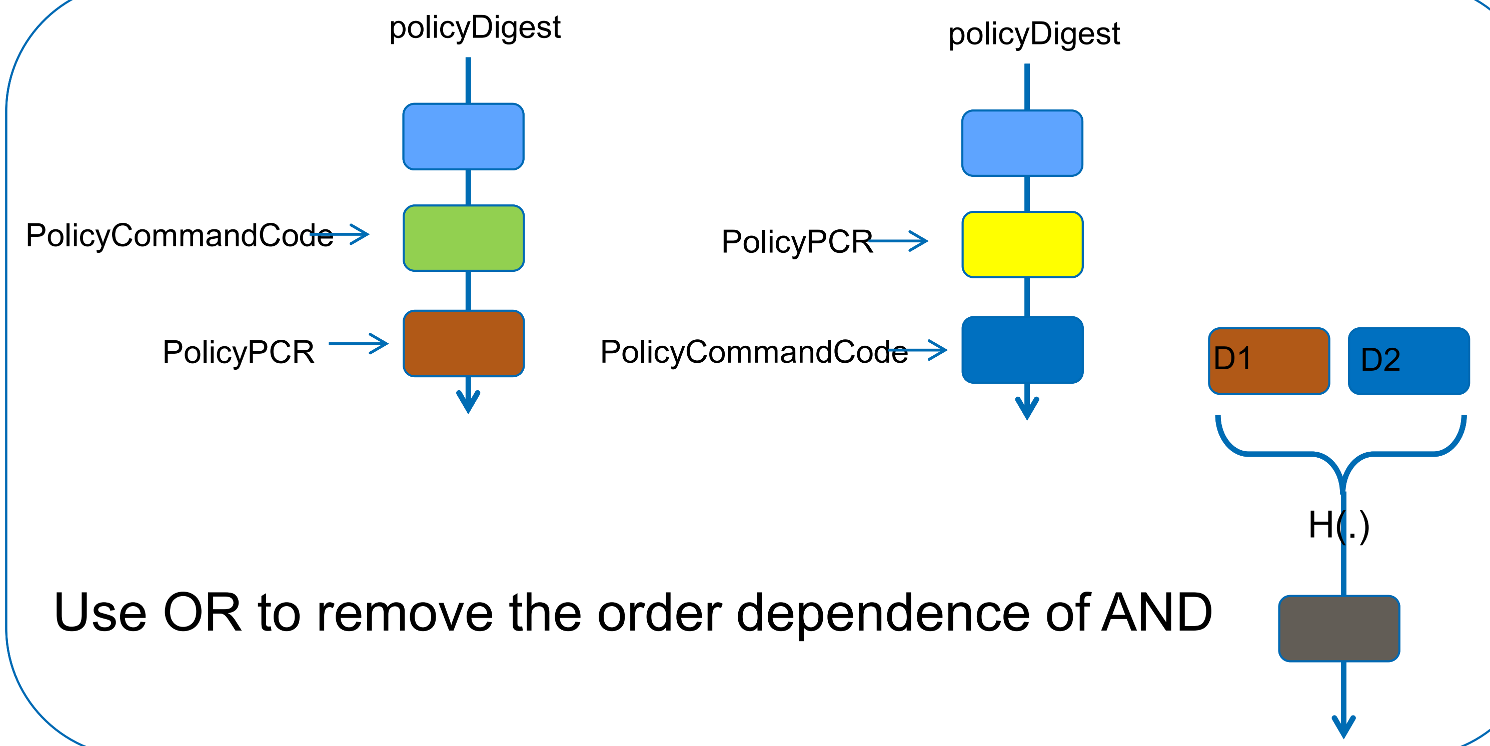


(Successful OR)



(Failing OR)

Policy conjunction



Use OR to remove the order dependence of AND

TPM2_PolicyAuthorize: Validate a signature on a policyDigest:

```
IF signature validates  AND signed text matches policySession->policyDigest
THEN
   newDigest :=  H(0 ‖ TPM2_CC_PolicyAuthorize ‖ H(pub) ‖ ...)
```

Remote attestation in general

Remote attestation in principle

```
HW = Samsung A52
OS = Android 11
App = Bank ID
```

Prover

Verifier

I'm talking to...

```
HW = Samsung A52
OS = Android 11
App = Bank ID
```

Binary attestation

Problem: What to attest?

First solution: attest a hash of the code running on the machine

- No ambiguity about which code is running
- Verifier needs to know hashes of every combination of valid software
  - Attestation needs to cover all code that affects the machine

Challenge: Number of hashes explodes as the number of software packages increases

- $N = n_1 \ n_2 \ \dots \ n_m$

Solutions:

- Limit number of software combinations (e.g. update all components together)
- Include list of installed software with attestation
  - Verifier only needs to know $N = n_1 + n_2 + \dots + n_m$ hashes

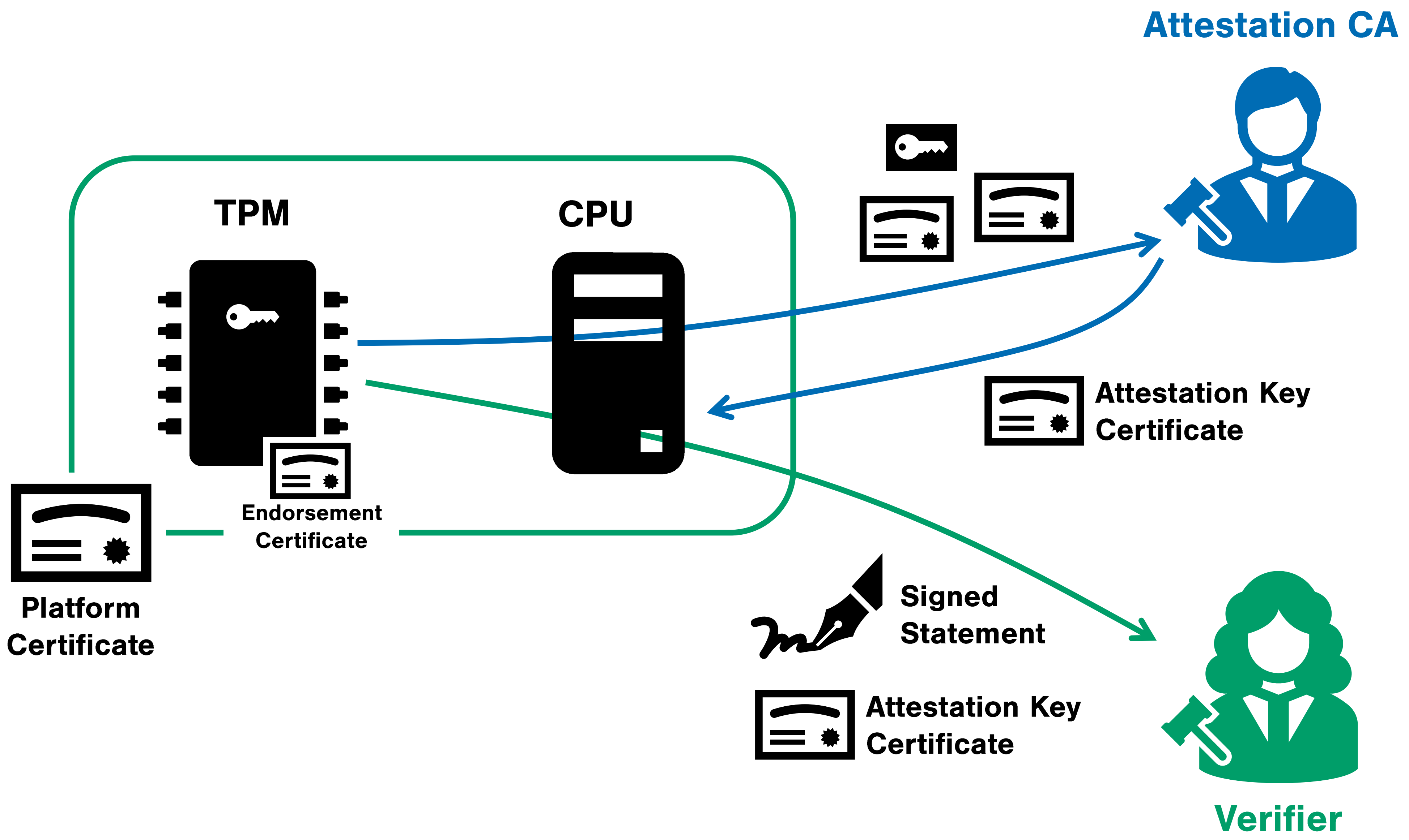
Property attestation

More properties are possible:

- Application was signed by a specific developer
- Application is in a specific state
- Particular computational result has some property

Remote attestation implementation

TPM remote attestation



TPM remote attestation types

Certification

- TPM promises that a key pair is protected by the TPM with certain properties
- “Key 51agca5613 is accessible by program mApp on OS mOS”
- TPM2_Certify() command

Quoting

- TPM promises that it is currently in some state
- “There is a platform running program mApp on OS mOS”
- TPM2_Quote() command

Measuring a Linux system

Easiest way: Linux Integrity Measurement Architecture

Kernel compares files read from disk with an aggregate measurement

- Aggregate measurement represents many files with one hash
- Can refuse access to the file if it doesn’t match (“appraisal”)

Aggregate measurement extended into a PCR

- Attested property: this system has a filesystem matching this measurement

We will talk about system integrity in greater detail next week

https://sourceforge.net/p/linux-ima/wiki/Home/

Remote attestation from secure boot

Recall: Secure boot only allows “correct” software to run

This can be used to provide remote attestation without a separate TPM

1. Key pair is stored in secure storage at manufacture time
   - Manufacturer certifies the public key
2. Device’s software is written to sign only true statements
3. Secure boot prevents other software from getting access to the key pair

You’ll learn about Trusted Execution Environments next week

- These help to make sure that #2 holds

Intel Software Guard eXtensions (SGX)

SGX provides enclaves: isolated applications protected from compromised software

- Protected even from the OS
- More about this next week

Two kinds of SGX attestation

- Local attestation: attestation between enclaves on the same machine
- Remote attestation: attestation from an enclave to a verifier on a different machine

Properties to attest:

- Enclave hash (`MRENCLAVE`)
- Enclave signer (`MRSIGNER`)
- Miscellaneous data (debug mode, etc.)
- Application-specific data

Contained in `sgx_report_t`

SGX Local Attestation

Enclave 1

```
EREPORT target, data ; Generate report
                     ; for enclave
                     ; target
```

```
MRSIGNER  = …
MRENCLAVE = …
DATA      = …
```

Enclave 2

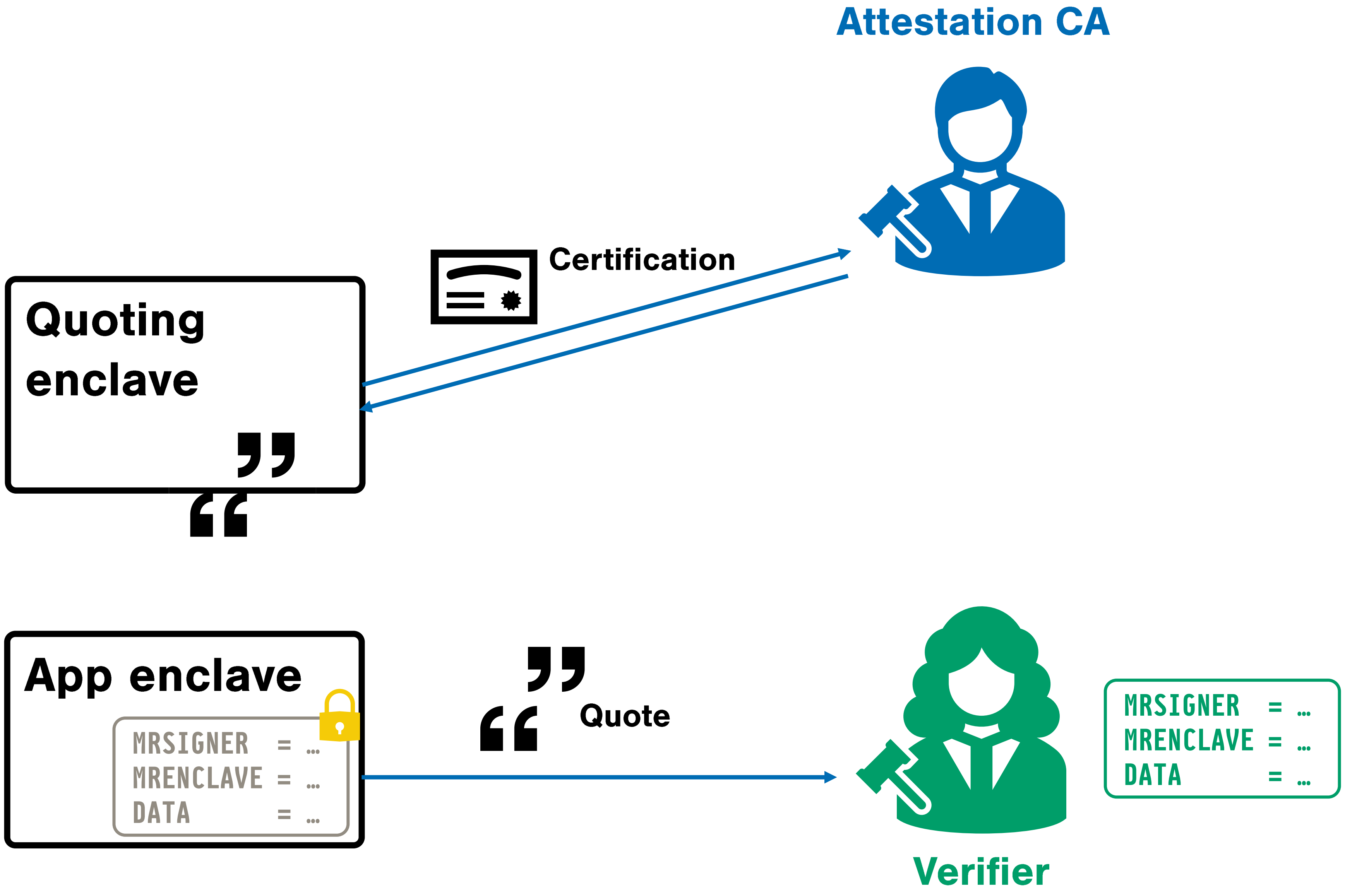
Attestation key

```
EGETKEY key_request ; Get this enclave's
                    ; key as described
                    ; by key_request
```

```
[verify report]
```

```
MRSIGNER  = …
MRENCLAVE = …
DATA      = …
```

SGX Remote Attestation



Recap

Booting a PC

- x86 boot process
- Secure boot
- Authenticated boot

Trusted platform module

- Measuring the system
- Authorisation

Remote attestation

- Types of attestation
- Different implementations