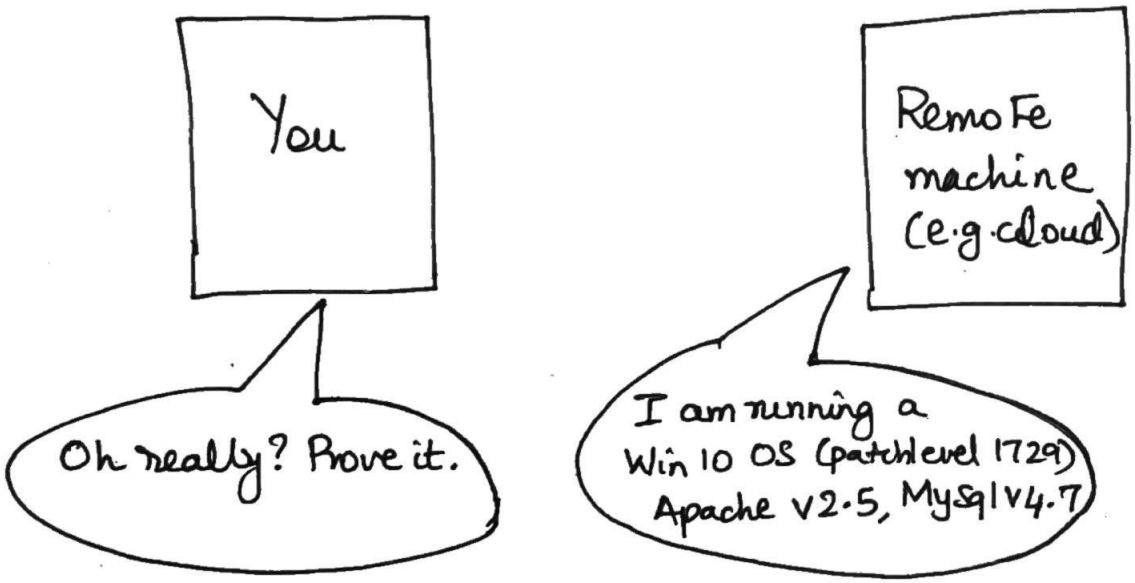
TRUSTED COMPUTING & ATTESTATION PROTOCOLS

PROBLEM: How does one attest the software stack running on a remote machine?



Attestation protocols are a pre-determined set of messages exchanged between the remote machine and your machine (in the picture above) that allow the remote machine to prove the integrity of its software stack to your machine.

APPLICATIONS OF ATTESTATION

(1) In doud computing, cloud client may wish to ensure that the cloud provider is indeed executing the software stack promised as part of the service-level agreement (SLA).

(2) Remote billing: Suppose a server charges the client based on the resources consumed by the client's code running on the server. Server produces a bill at the end. How does client trust that the bill is correct?

One solution: Client and server together agree on the billing software that the server will run. (Thus, client trusts the output of the billing software). Client only needs to establish that the server is running that billing software.

client can ask the server to attest its software stack & ensure that billing software is running

APPLICATIONS (CONTINUED...)

③ MALWARE DETECTION / PROVING ABSENCE OF MALWARE.

Client may wish to ensure that server is only running known good software as part of the software stack. Client can refuse to give its business to the server otherwise.

Client studies the attestation report (also called a quote) to ensure the above property.

④ CORPORATE SETTINGS / ENTERPRISE MOBILE DEVICES.

Increasing trend in enterprises to allow employees to bring their own devices (phone / laptop) etc to work. Company provides a VM setup for the work, but hardware belongs to employee.

BYOD: Bring your own device.

How can enterprise ensure that the employee's device is free of software that could potentially be used to steal the company's corporate assets? (e.g. code, internal documents, etc).

Relevant in remote work (a.k.a "work from home")

For all this to be possible, the client needs to have some root of trust on the server's side. Failing the existence of that root of trust, attestation is impossible.

That root of trust is called trusted hardware.
Trusted hardware comes in many forms nowadays.

- TPM (Trusted Platform Module) — the first widely deployed, commercial trusted hardware mechanism. Your laptop likely has it (check the device drivers). Our focus today.

- SGX (Software Guard Extensions) from Intel. Allowed encrypted execution on the cloud. Attestation is an important part of SGX. We will be studying SGX too.

- ARM TrustZone. Trusted hardware on mobile devices. Forms the basis of Samsung Knox Enterprise solutions. Can protect mobile payment apps. Likely there on your phone today.

Today: TPM.

- Proposed in early 2000s.
- Now in most modern chipsets (costs roughly $5 or so)
- Provides a root of trust on the machine and serves as the basis of all attestation protocols.
- The full specification of TPM has way more detail but we're only going to be looking at the core ideas in an abstract way.

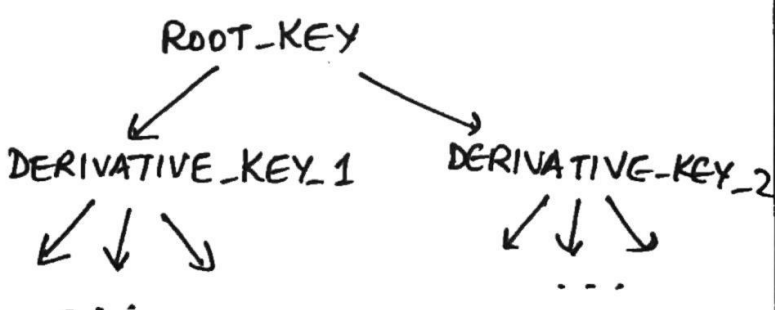
So, what is the TPM?

(★) A piece of tamper-resistant hardware.
→ if you try physical attacks (e.g., open the chip packaging to read on-board contents) it destructs.

(★) With a public/private key pair.
→ Private key hardwired by manufacturer.
→ Public key given to you on purchase (certified, of course).

(★) Capable of performing crypto operations.
→ (Simple RSA encryption & digital signatures).

- Stores data securely in on-chip registers.
  - Platform-configuration register or PCR. About 10 or so. We'll just abstract to 1.
  - on-board, secure storage.
- Supports an API called tpm_extend.
  - "extends" the content of the PCR register with a supplied value. More in the next slide.
- Has a complex key hierarchy:
  - → but we're not going to be concerned with the key hierarchy in this course.
  - ↳ key hierarchy means:



→ Root key is the manufacturer hardwired key.

→ Never used directly for encryption/signatures.

→ But that is what we'll pretend is used when we study the protocol.

TPM_EXTEND

Inputs:

① an input value v, typically a hash value computed outside the TPM

② PCR register number

(so, let's say current value of this register is cure_PCR_value

Functionality: New_PCR_value = SHA-256(cure_PCR_value || v).

We will pretend there is only one PCR register (for simplicity). Thus, our API is

tpm_extend (v)

If value currently in the PCR is u, then after tpm_extend, the value becomes

SHA-256 (u || v).

Suppose the value of the PCR to start with is 0, what is its value after invoking tpm_extend with three arguments x, y, z?

- After step 1: SHA256 (0||x)
- After step 2: SHA256 (SHA256 (0||x) ||y)
- After step 3: SHA256 (SHA256 (SHA256 (0||x) ||y) ||z)

This nested structure is also called a hash-chain

[Yes, this does sound related to and is indeed related to what goes on in a block-chain]

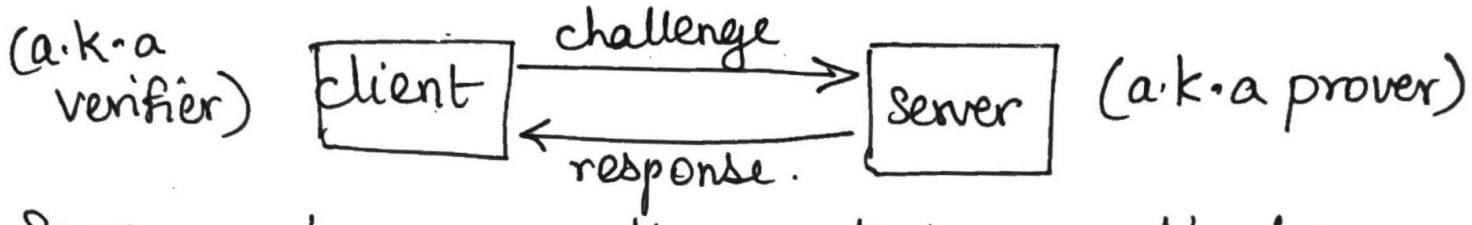
- Note that x, y, & z are themselves usually hash values. Say $x = SHA(v_1)$, $y = SHA(v_2)$ & $z = SHA(v_3)$

Then, the value after step3 is

SHA256 (SHA256 (SHA256 (0||SHA256 (v_1)) ||SHA256 (v_2)) ||
SHA256 (v_3)))

Note that SHA256 ~~values~~ is a function that cannot be reversed easily.

So, given all this, how does attestation work?



Server must run a software stack amenable for attestation.

We will look at one particular type of attestation, i.e., trusted boot / secure boot.

At boot time, server gives snapshot of its software stack to client, who can verify it.

How does a machine boot?

- BIOS → Bootloader → OS image → Applications.

The server must declare upfront the versions of the entire boot chain to the client. This is part of the attestation report.

When the TPM boots, the PCR is initialized to a known value (say, 0)

- When hardware invokes the BIOS, it "measures" the BIOS code, i.e, computes a SHA256 hash of the code of the BIOS & invokes

$$\xi \text{tpm\_extend}(\text{SHA256}(\text{BIOS}))$$

$$v_1 = \text{SHA256}(0 \,||\, \text{SHA256}(\text{BIOS}))$$

- BIOS in turn "measures" bootloader code before invoking it.

$$\text{tpm\_extend}(\text{SHA256}(\text{bootloader}))$$

$$v_2 = \text{SHA256}(v_1 \,||\, \text{SHA256}(\text{bootloader}))$$

- Bootloader measures the OS code.

$$\text{tpm\_extend}(\text{SHA256}(\text{OS}))$$

$$v_3 = \text{SHA256}(v_2 \,||\, \text{SHA256}(\text{OS}))$$

- OS in turn measures applications that start.
  - → But this is not "deterministic" like boot as apps can start in any order.
  - → OS keeps a log "measurement list" to keep track of this order.
  - → See assigned reading for details **IMPORTANT**

- Let's ignore applications for now (but see the paper for details) an just focus on boot.
- What is the PCR value at the end of the boot? v_3.
- If the server booted as promised, and if the software stack (BIOS, Bootloader, OS) are not corrupted, the value in the PCR of the server must be v_3.
- If anything is corrupted, value in PCR $\neq v_3$.
- There is a chain of trust we've built.

  → HARDWARE says BIOS sha is SHA256(BIOS)

  → BIOS says Bootloader's sha is SHA256(Bootloader)

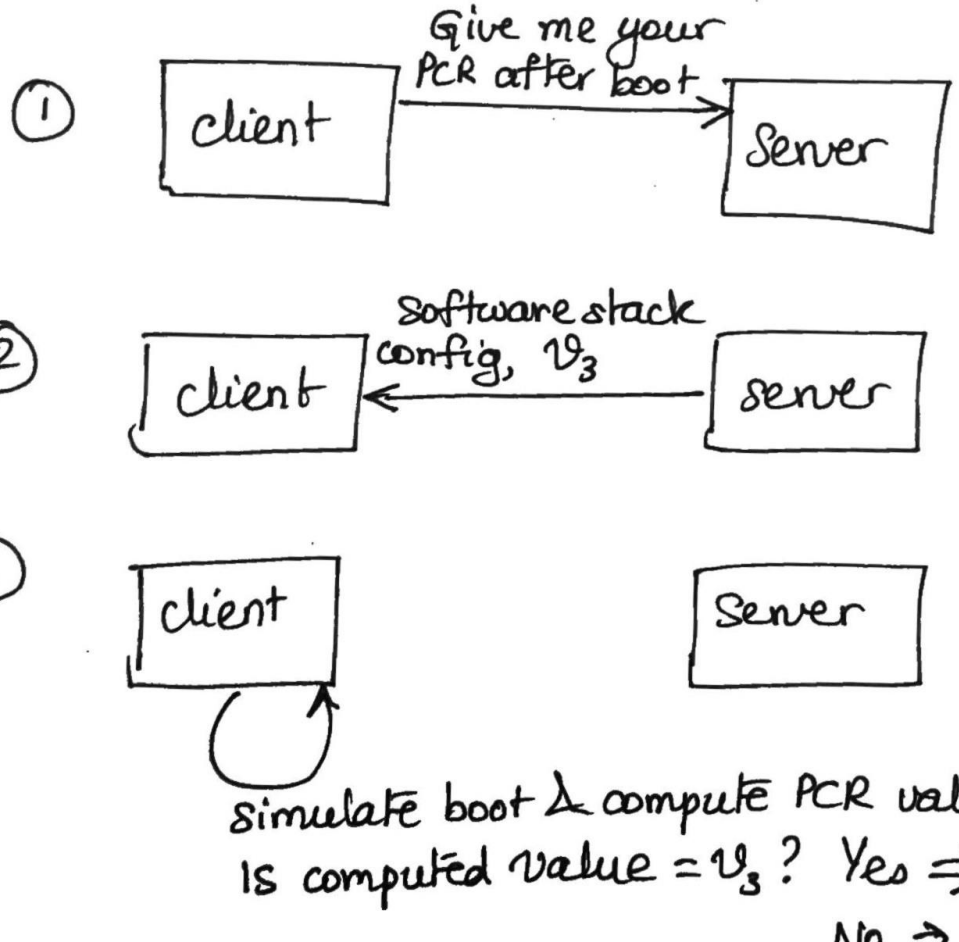
  → Bootloader says OS code sha is SHA256(OS)

Each hash computed by prior layer.
And the end, or "root" of this chain of trust is the hardware, which must be tamper-proof.
Hence the name <u>trusted hardware</u>

- Importantly, if server declares its software stack to client, client can "simulate" the boot process of the server, and compute what value *should* be in the server's PCR. Why?

  A: It is just a series of SHA256 computations.

- So:



Can anyone see a problem?

Replay attacks! Server can always replay v_3.
How to prevent replay? With nonces.

So:

(1) client —— Give me your PCR after boot, Nonce = r. ——→ server

(2) client

server

r is passed to TPM, and TPM is asked to produce a "quote" of the value in its PCR. TPM computes Digital sig $(r||PCR)$ ie $E_{TPM_{priv}}(r||PCR) \rightarrow$ digsig

(3) client ←—— Software stack config, PCR_value, dig-sig. —— server

(4) client

simulate boot, get value of PCR, check for a match. & verify the dig sig with TPM_{pub} (decrypt the sig with TPM_{pub})

X