Hardware Root Of Trust

Chapter 5

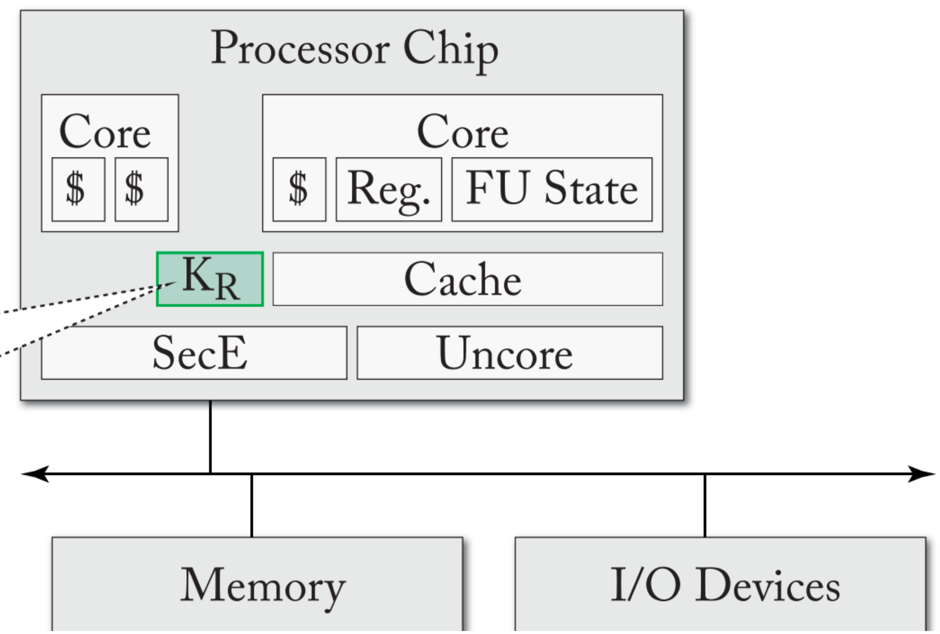
[1] J. Szefer, “Principles of secure processor architecture design,” *Synth. Lect. Comput. Archit.*, vol. 13, no. 3, pp. 1–173, 2018.

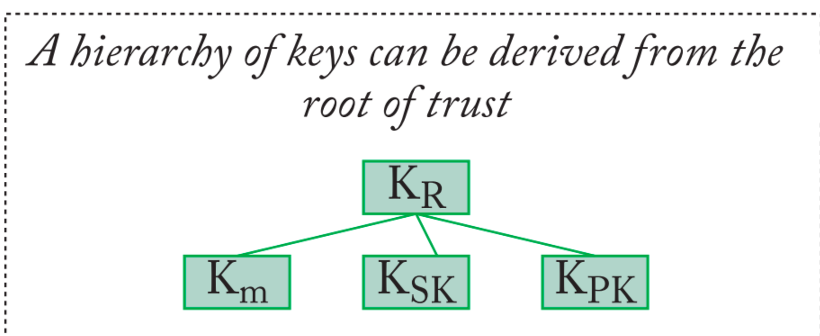
The Root of Trust

- Measure the chain of trust
  - Trusted authenticated boot, remote attestation, and data sealing
  - Runtime attestation and continuous attestation
- Secure processor should ensure for TEE
  - **Authentication**: the processor should authenticate himself (to guarantee that the information coming from it can be trusted).
    - E.g., deriving asymmetric crypto keys from a manufacture dependent unique secret key
  - **Confidentiality**: data going out of processor should be protected.
    - E.g., For memory using a key created at boot time (ephemeral) for persistent keys are persistent and generated (in a systematic way)
  - **Integrity**: additional keys are required for integrity checking. Cryptographic hashing (one-way) from secret key to prevent tampering (e.g., replays)

The Processor Key (The root of trust)

- **Kr** is a unique key per chip: is the root of trust. Others derived from that with one-way hashing
- **Never should leave the chip boundary**
- Created at manufacturing time (e.g., eFUSE)
- Manufacturer **shouldn't** hold a copy of the key
- Use PUF?, more costly to guarantee reliability/stability



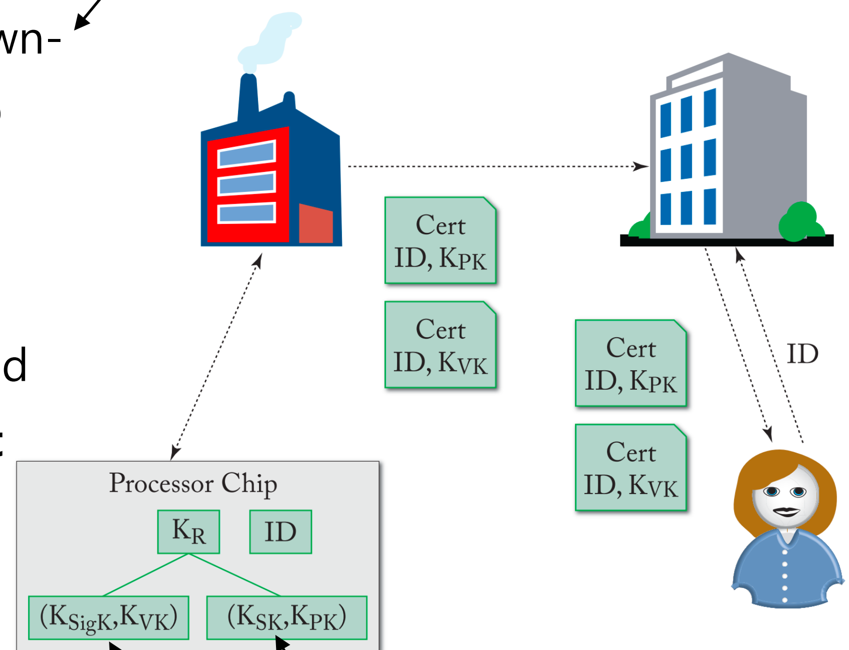


Keys belong to TC, if TEE need keys
, they will be generated from them

PKI and Secure Processors

- **How to tell if a processor is who it says he is?**
- PKI is required to distributed the certificate of the public key of the processors
- The manufacturer should keep a list of known-good keys and the associated processors ID
- **K_r** is used to derive different pairs for **encryption** (K_{sk}, K_{pk}) or **signing** (K_{SigK}, K_{vk})
- ID, public (K_{pk}) and verification (K_{vk}) keys, and corresponding certificates, are **generated at manufacture** and given to the certificate authority
- Given some ID of a processor, **the users can obtain from CA the certificates for that processor's keys**

Certificate revocation is an open issue (e.g., a Kr compromised)



For authentication

For confidentiality, integrity (i.e,, TCB)

Aside: RSA and Asymmetric Cryptography

1. Generate two large random primes, p and q, of approximately equal size such that their product $n=pq$ is of the required bit length, e.g., 1024 bits.
2. Compute $n=pq$ and $\phi=(p-1)(q-1)$
3. Choose an integer e, $1<e<\phi$, such that $\gcd(e,\phi)=1$
4. Compute the secret exponent d, $1<d<\phi$, such that $ed\equiv 1 \bmod \phi$
5. The public key is (e,n) and the private key (d,n). Keep other values secret.

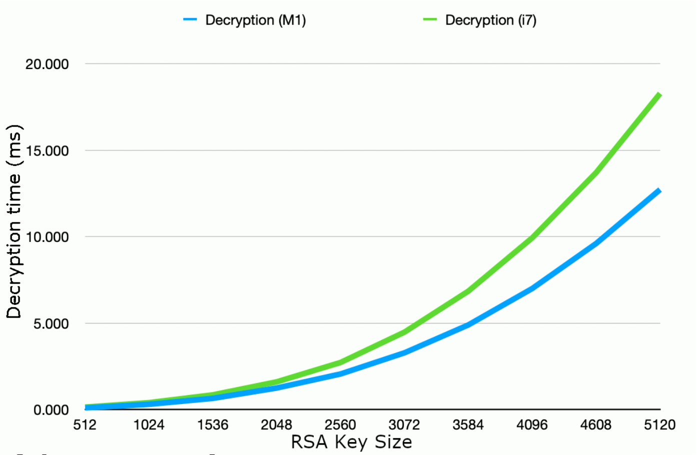
- n is known as the *modulus*.
- e is known as the ***public*** *exponent* or ***encryption*** *exponent* or just the *exponent*.
- d is known as the ***secret*** *exponent* or ***decryption*** *exponent*.

Aside: RSA Example

- Choose $p = 3$ *and* $q = 11$
- Compute $n = p * q = 3 * 11 = 33$
- Compute φ (n) = (p - 1) * (q - 1) = 2 * 10 = 20
- Choose e such that 1 < e < ϕ (n) and e and ϕ (n) are coprime. Let $e = 7$
- Compute a value for d such that (d * e) % ϕ(n) = 1. One solution is $d = 3$ $[(3 * 7) \% 20 = 1]$
- Public key is (e, n) => (7, 33)
- Private key is (d, n) => (3, 33)
- The encryption of $m = 2$ is $c = \mathbf{2^{7}} \% 33 = 29$
- The decryption of $c = 29$ is $m = \mathbf{29^{3}} \% 33 = 2$

Aside: Computational Cost of RSA

- Public key operations $O(n^2)$, Private Key operations $O(n^3)$
  - Very costly when size of the key increases

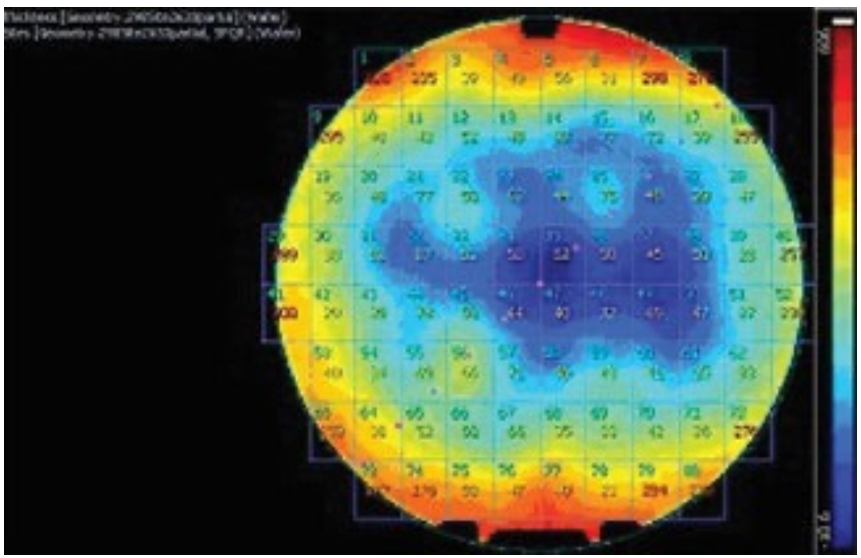


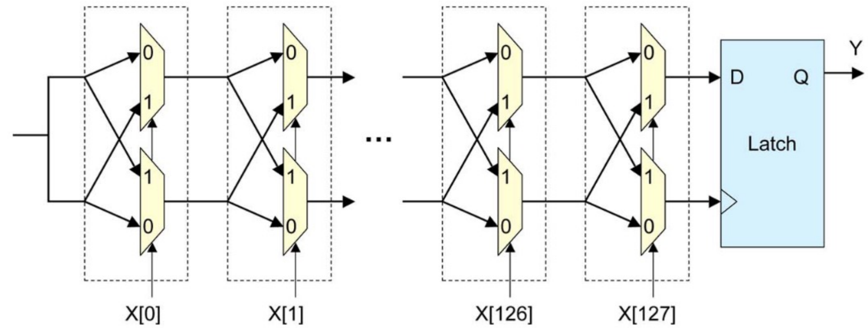
- Many *A*B mod N* operations
  - No support in current processors but SIMD (e.g., AVX) might help
  - External hardware support; e.g., PCI Cards, FPGA, …

- ECDSA [Elliptic Curve Cryptography] is replacing RSA everywhere (224-bit ECDSA is equivalent to 2048-bit RSA)

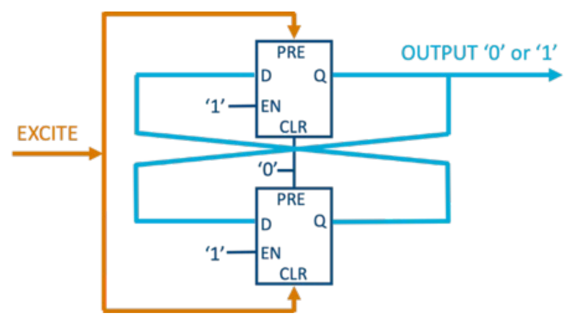
Aside: Physically Unclonable functions (PUF)

- Circuits that can't be replicated because depends upon small variabilities during the manufacturing process (die-to-die variability)
  - Mainly random processes across wafer





Arbiter PUF:
Which path is faster determining the output for the challenge (X)



Butterfly PUF:
Excitation in preset and clr creates instability in cross coupled inputs but settles in a stable output

Access to the Root Of Trust

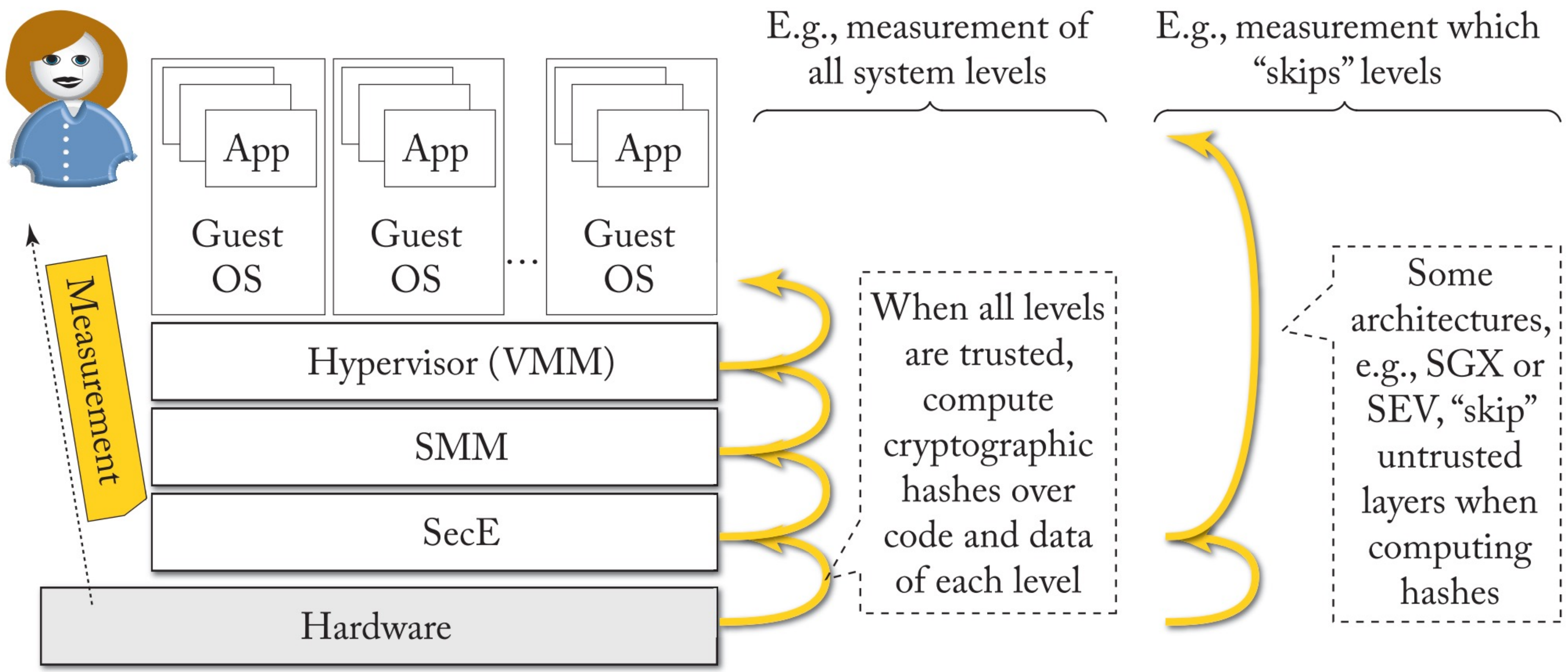
- Although complex physical attacks and/or rogue manufacturer can compromise the Root of Trust, the most likely scenario is **bugs in TCB**
  - Low level code (e.g., UEFI) should be able to access secret keys (e.g., to sign digital messages generated by the processor)
  - Code is kept in flash memory
  - Bugs in that code can lead to exploits that can leak the secret key

Chain of Trust and Measurements

- When run TEE, is crucial to be correctly configured
  - A trusted small BIOS loaded at system boot (first thing to load!)
  - This piece can "**measure**" next piece to be loaded (e.g., firmware)
  - Firmware "**measure**" the OS that starts next
  - …
- **Measure ==** act of generating a cryptographic hash of the code (and data) before it is executed
  - If **measurement != expectations**, tampering is detected
- Measurements can be chained (or completed) from multiple components.
  - Each measurement is "**extend**" with the next one in the chain (coined by TPM)
  - This idea is used by Trusted Platform Module (TPM): low cost dedicated to providing root of trust. Protects against software attacks (mainly rootkits), secure disk cyphering, machine authentication (e.g., windows hello)

Chain of Trust and Measurements

- Chains of trust can include untrusted levels
- **Usually**, only trusted components are measured
  - **Only if software is part of TCB and TEE is measured**



Trusted and Authenticated Boot

- **Trusted boot** means system might boots only if all measurements are correct

- **Authenticated boot** means users can access to all measurements taken during boot (but always boot if aren't correct)

- In **trusted boot** If measurements aren't correct no standardized actions
  - A system can stop executing or secret data won't be decrypted
  - But what if the system is part of a critical infrastructure?
  - But what is my laptop and I want to prevent Linux + passwd cracker to unencrypt my disk content?

Measurement Validation and Remote Attestation

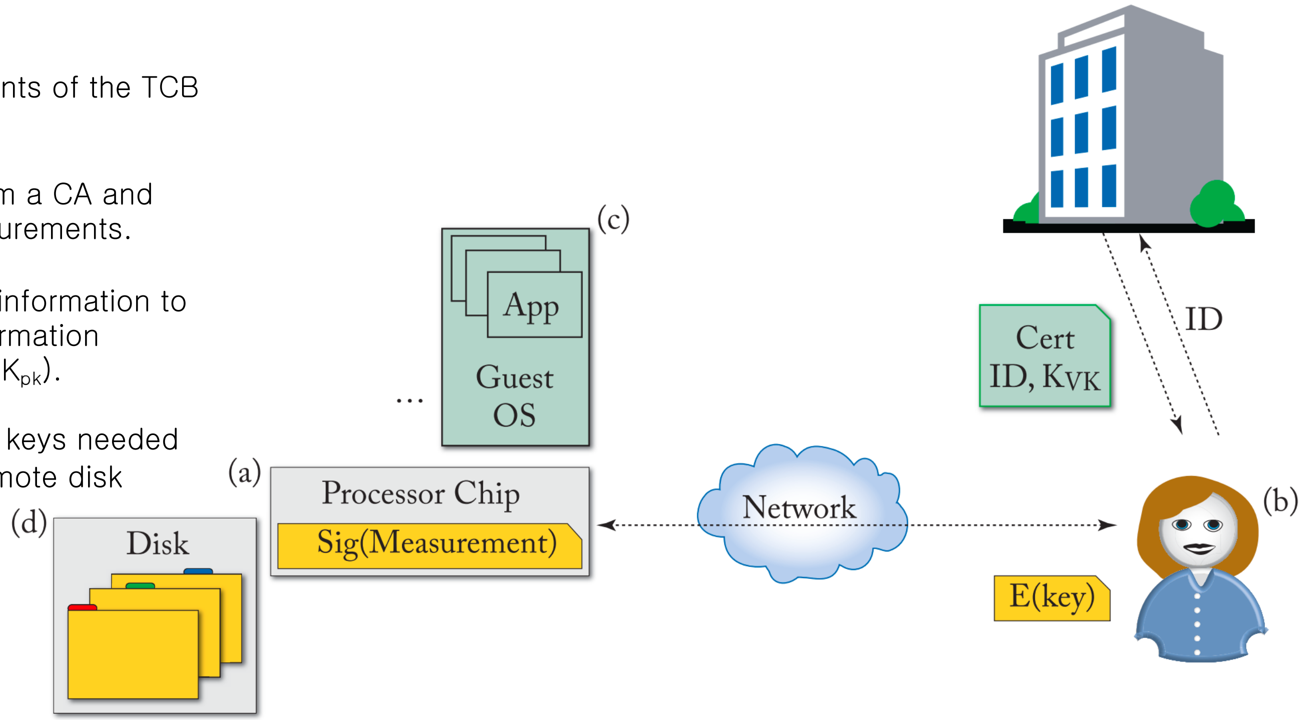
- Main method: check measurements against a list of **good values**
  - But any change in the boot (e.g., change in the version of the loader) might completely change the resulting hash ➔ Can't have all combinations of system software locally "tagged": Use PKI for key distribution

(a) Processor generate measurements of the TCB (and TEE) signed with K_r.

(b) User can obtain certificates from a CA and validate the signatures of the measurements.

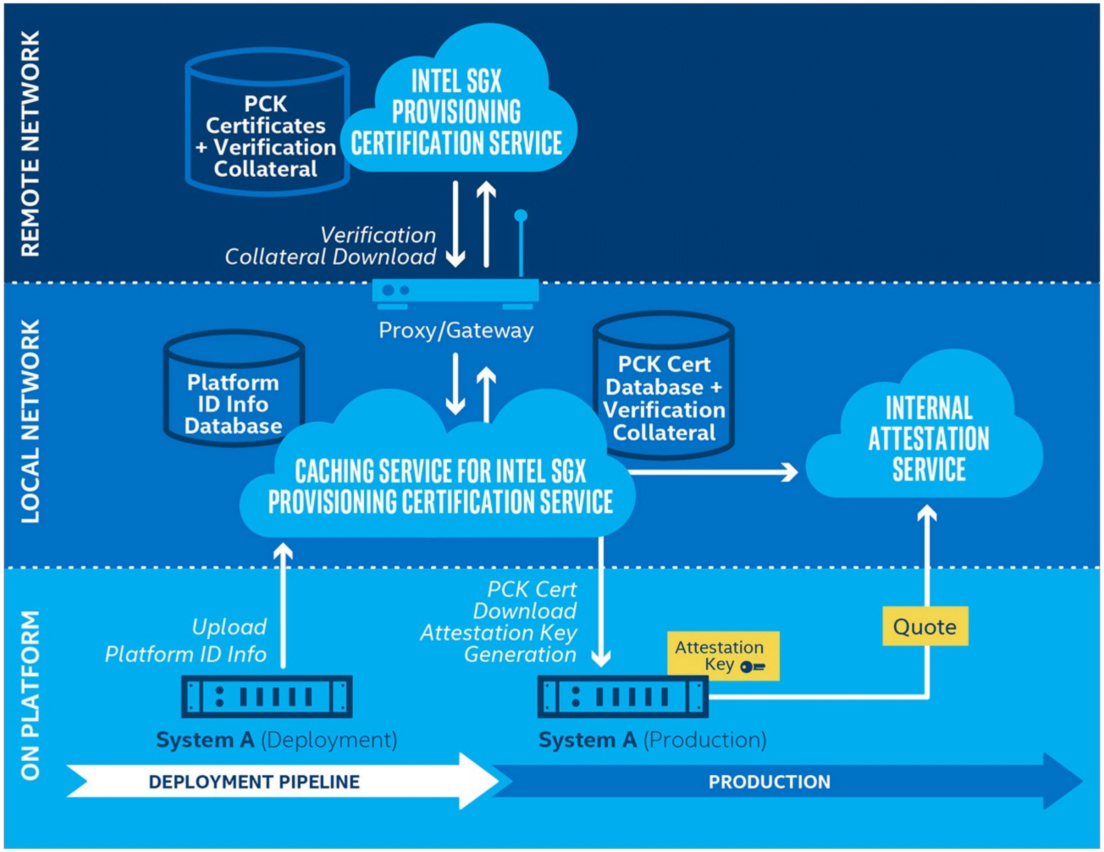
(c) User can send some sensitive information to the TEE, by sending encrypted information (and decryption key protected with K_{pk}).

(d) The user can also send to TEE keys needed for decryption of data stored on remote disk



Example: Remote Attestation with Intel SGX

- Allows a hardware entity or a combination of hardware and software to gain a remote provider's (also known as the relying party) or producer's trust.

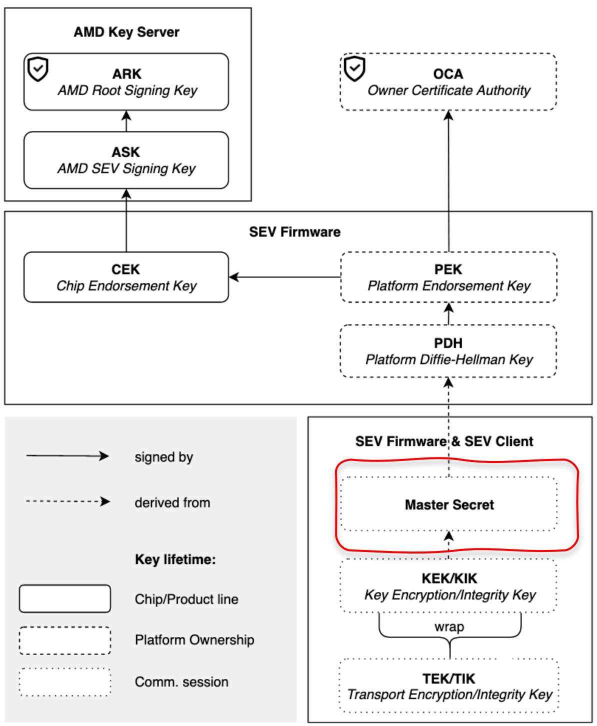


- **Intel ME** Core iX 11xxx<
- **Intel SPS** All Xeon >D >2020
- (Server Platform Services)

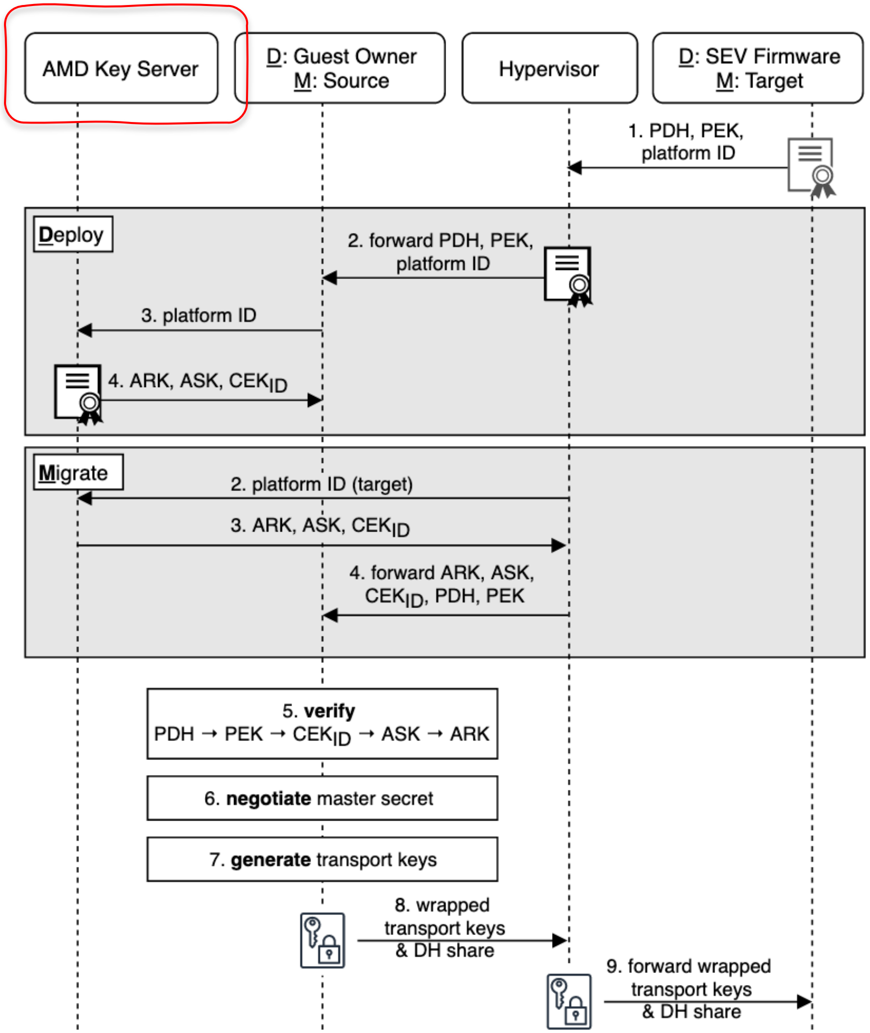
Example: Remote Attestation with AMD SEV

- https://arxiv.org/pdf/1908.11680.pdf

Cryptographic keys in SEV.



SEV Secure Chanel (M: migration, D; deploy)



Sealing

- Act of **encrypting** some data with a key derived from a measurement.
- Unsealing, is the act of decrypting data with a key derived form a measurement.
- Tie the data to a specific hardware and software configuration.
  - For example, if **hard drive data is sealed at shutdown (local or remote)**, then next time the system boots up, if there is any change to the measurements (e.g., any of the trusted software was somehow **changed**), then the proper decryption key **cannot be derived**, and data cannot be decrypted.
- This prevents malicious uses from accessing data if they changed the software while the system was turned off (e.g., **replacing the boot disk**)

Time-to-check to time-to-use attacks

- Measurement technique does not say anything about the code **after it was measured**.
  - It only is based on raw measurements of static code as it was right before the piece of software executed.
  - This leads to well-known class of Time-of-Check to Time-of-Use (TOC-TOU) problems

- The system may be **correct at time t_0**, but it may become **compromised at time t_1**
  - if user **asks for the values of measurements at some later time t_2**, he or she will get the hash value that states that system was okay at t_0, but no information about state at t_2 is gained

- One solution to this problem is to perform **continuous attestation** at system runtime.

Runtime Attestation and Continuous Monitoring of TCB and TEEs

- Monitoring of the execution of the system via co-processor
  - Validates continuously the execution path (combined with PMU)

- Observing the execution pattern (Basic Block size, branch predictor accuracy, performance metric …) we can infer when a program deviates from expected behavior
  - Mostly applied to untrusted component (Use in TCB not explored)

- Using Control Flow Graph of the program (from the source code of the program) to check if there are deviations (valid only for software, not clear for hardware)

- Using PMU to detect anomalies can be **easily defeated** by hiding the attack "in the noise" (e.g., by carefully modifying slightly the normal execution characteristics)

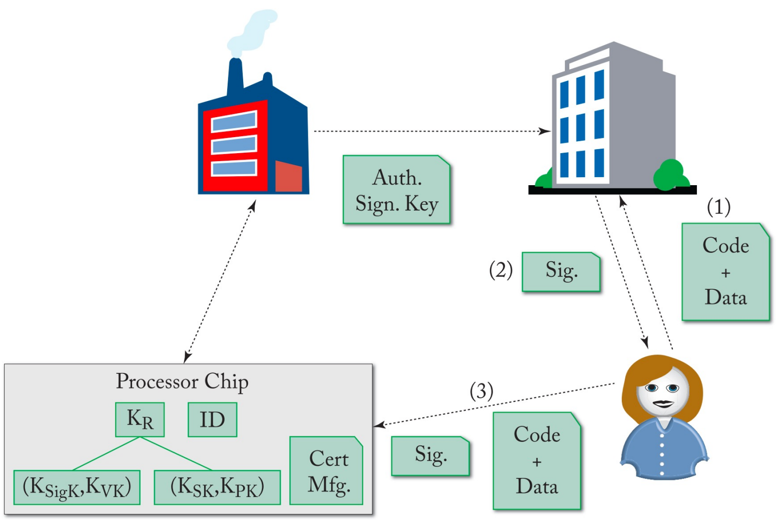
PUFs and Root of Trust

- PUFs are impossible to replicate (even for the manufacturer) and provides a good fingerprint for the systems
  - Turns process variation problem into a virtue

- Can permanently **bind hardware-software** by condition TEE execution path (i.e., not only in measurements)
  - The PUF might generate a stream of outputs for the TEE
    - E.g., Generate a seed for a PRNG
    - E.g., Use for some form of indirection (DRAM PUF)

Limiting the Execution to Only Authorized Code

- TCB updates (or protected) software can be authenticated with a separate key (manufacturer dependent)
- Add public key to the processor at manufacture time
- Only software signed by manufacturer can be sent to the processor
- Code can be **signed under demand** using also processor keys (e.g. to prevent install certain firmware out of a particular time window)



Lock-In and Privacy Concerns

- User depends on manufacturer to send information to TEE
  - E.g. Software running inside SGX requires Intel signature
- When TCB and TEE is authenticated, use processor specify keys that can be used by the Certification Authority to infer information about the systems requesting the update
- Direct Anonymous Attestation (DAA) on TPM addresses these issues (partially)
  - Camenisch-Lysyanskaya Signatures

Root of Trust Assumptions

- Unique key assumptions
  - Manufacturer can never reuse the same key or use PUF
- Protected root of trust assumption
  - Kr will never be disclosed
  - TCB never will leak it
- Fresh measurement assumption
  - Need to be fresh to prevent TOC-TOU attacks

Example Attack: AMD TCB exploit for attestation

- **One Glitch to Rule Them All (2021)** https://arxiv.org/abs/2108.04575
  - https://github.com/PSPReverse/amd-sev-migration-attack
  - Voltage glitching attack that allows an attacker to execute custom payloads on the AMD-SPs --> Extract keys to deploy "custom" firmware
    - Allow to access secured VM using live-migration
  - Allows to generate false attestation reports (in the "custom" firmware server) (because we can change root of trust and fake the secure protocol)
    - An attacker without physical access to the target host can use extracted endorsement keys to attack SEV-protected VM s

Example Attack : SGX Attacks

- https://arxiv.org/pdf/2006.13598.pdf

				Impact						Mitigations			
Attacks (abbreviated titles)	Type	SGX Specific	Targeted attack	Page access pattern	Instruction trace	Instruction latency	Memory access pattern	Memory Contents	Fault Injection	Microcode patch	System design	Compiler/SDK	Application design
Controlled-Channel [9]	sec. III-A	●	●	●	○	○	○	○	○	○	○	●[52]	●[55]
Stealthy Page Table [10]	sec. III-A	●	●	●	○	○	○	○	○	○	○	●[52]	○
SGX-Step [11]	sec. III-A	●	○	●	●	●	○	○	○	○	○	●[11] [48]	○
Nemesis [12]	sec. III-A	●	●	○	○	●	○	○	○	○	○	●[50]	○
Off Limits [13]	sec. III-A	●	●	●	◐	○	◐	○	○	●[13]	○	○	●[13]
Leaky Cauldron [14]	sec. III-B	●	●	●	○	○	●	○	○	○	○	●[50] [51]	○
CacheZoom [20]	sec. III-B	●	●	○	○	○	●	○	○	○	○	●[50] [51]	●[56]
Cache Attacks on SGX [21]	sec. III-B	●	●	○	○	○	●	○	○	○	○	○	●[56]
Malware Guard Extensions [22]	sec. III-B	●	●	○	○	○	●	○	○	○	●[22]	○	○
Software Grand Exposure [23]	sec. III-B	●	●	○	○	○	●	○	○	○	○	○	●[23]
CacheQuote [24]	sec. III-B	●	●	○	○	○	●	○	○	○	●[24]	○	○
MemJam [25]	sec. III-B	○	●	○	○	○	●	○	○	○	○	●[49]	○
Branch Shadowing [26]	sec. III-C	●	●	○	●	○	○	○	○	○	○	●[26] [48]	○
BranchScope [27]	sec. III-C	○	●	○	●	○	○	○	○	○	○	●[27] [48]	○
Bluethunder [28]	sec. III-C	●	●	○	●	○	○	○	○	○	○	●[28]	○
SgxPectre [31]	sec. III-D	●	●	○	○	○	○	●	○	●[47]	○	○	○
SpectreRSB [32]	sec. III-D	○	●	○	○	○	○	●	○	○	○	○	●[32]
Spectre v1 [33]	sec. III-D	○	●	○	○	○	○	●	○	●[47]	○	○	○
Foreshadow-SGX [34]	sec. III-E	●	○	○	○	○	○	●	○	●[35]	○	○	○
RIDL [38]	sec. III-F	○	●	○	○	○	○	●	○	●[38]	○	○	○
ZombieLoad [39]	sec. III-F	○	●	○	○	○	○	●	○	●[39]	○	○	○
CacheOut [40]	sec. III-F	●	●	○	○	○	○	●	○	◐[40]	○	○	○
CrossTalk [41]	sec. III-F	●	○	○	○	○	○	●	○	●[41]	○	○	○
Plundervolt [44]	sec. III-G	●	◐	○	○	○	○	○	●	●[44]	○	○	○