OS Security and Trust

Raj Kane
3/14/22

Topics

- What is OS security? Trusted computing?
- Reference monitor and access control
- TPMs
- Digital signatures and attestation

OS Security Properties

CIA properties *[5]*:

- **C**onfidentiality: "preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information"
- **I**ntegrity: "guarding against improper information modification or destruction, and includes ensuring information **nonrepudiation** and **authenticity**"
- **A**vailability: "ensuring timely and reliable access to and use of information"

Authenticity: "property that data originated from its purported source" -- *[4]*

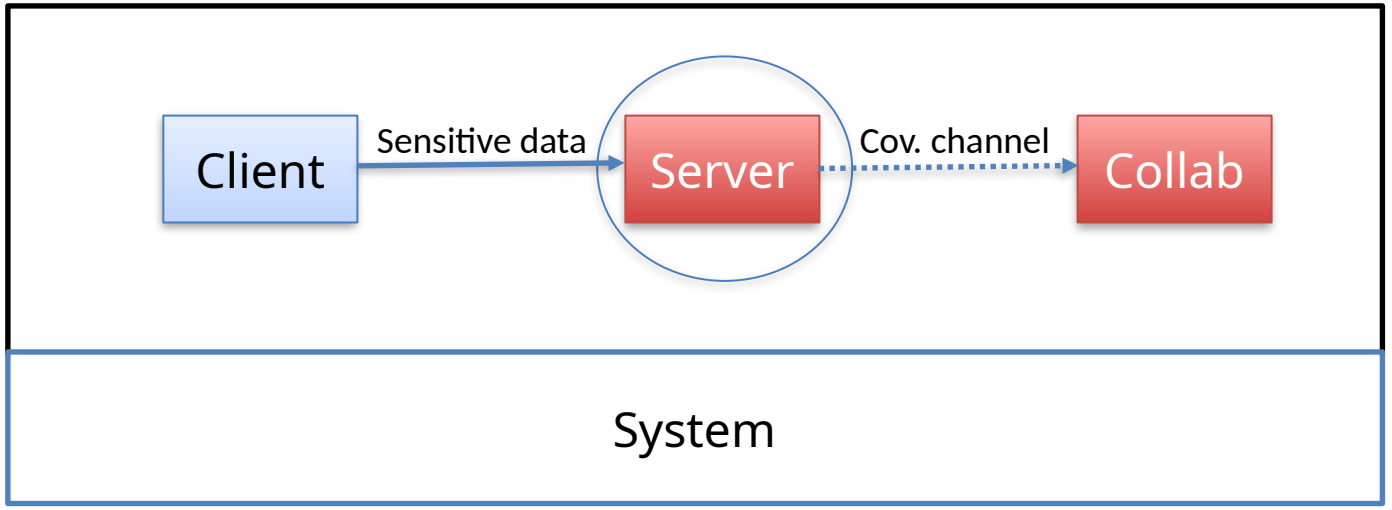
Nonrepudiation: "Assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the information." -- *[7]*

Additional definitions at *https://csrc.nist.gov/glossary*

Is a Truly Secure OS Possible?

Extremely challenging in practice.

- Rise of software complexity
- Inertia in updating one's system
- Confinement cannot prevent covert channels



Trust in a System

Systems should behave as expected by enforcing their security policies. Trusted components are a prerequisite for security.

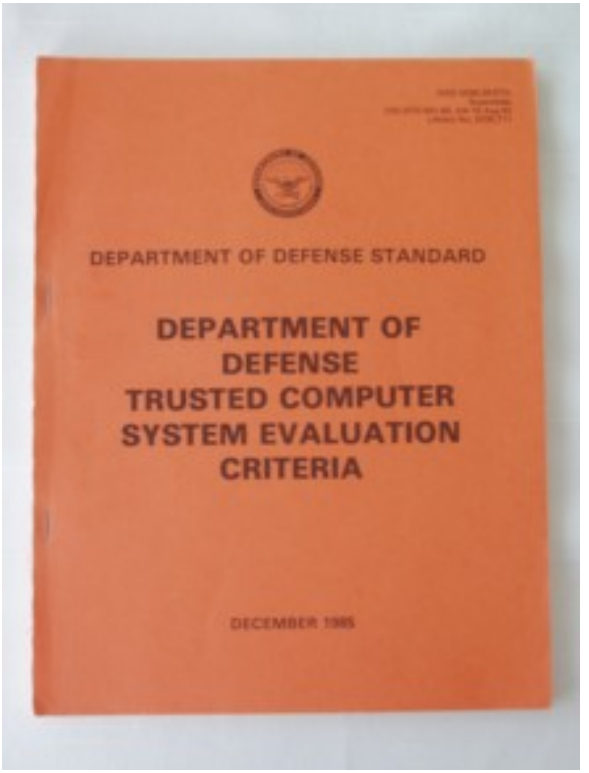
Trusted components may themselves be vulnerable. How can we verify that the trust placed in them is justified, so that the system is not jeopardized?

Can we limit the amount of trust needed?

Trusted Component

"a ***trusted*** system or component is defined as one whose failure can break the security policy; and a ***trustworthy*** system or component is defined as one that will not fail." -- *Orange Book [1]*

"A ***trusted*** system or component is one that behaves in the expected manner for a particular purpose." -- *Trusted Computing Group*



Trusted Computing Group

AMD

intel

Microsoft

IBM

infineon

And more…

TRUSTED COMPUTING GROUP™

"Through open standards and specifications, Trusted Computing Group (TCG) enables secure computing."

"Virtually all enterprise PCs, many servers and embedded systems include the TPM; while networking equipment, drives and other devices and systems deploy other TCG specifications, including self-encrypting drives and network security specifications." -- *https://trustedcomputinggroup.org/about*

Trusted Computing Base

TCB = subset of system components that *have* to be correct to enforce security policies

- May include: hardware, TPM, kernel, privileged programs (e.g. SETUID programs)

Desirable properties:

- Small
- Auditable

Reference monitor: one place to mediate all accesses to TCB

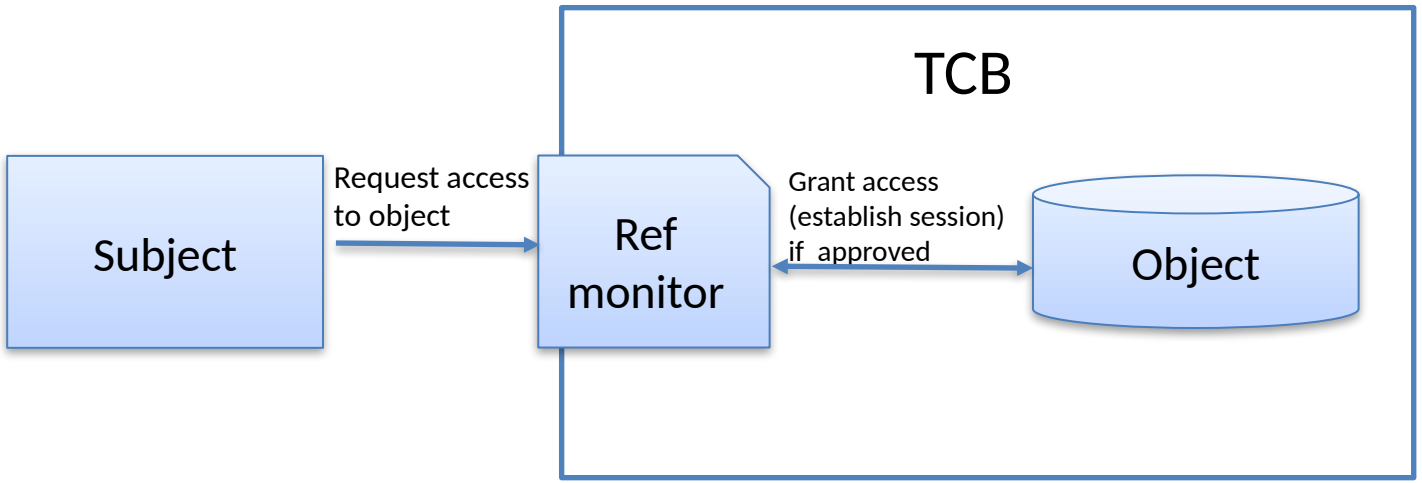
Reference Monitor

Reference monitor is an access control concept, implemented as a module through which all accesses to the rest of the TCB are routed.

- Claim: yields correct enforcement of access control policies
- Issues: Time of Check to Time of Use (race conditions), covert channels

Ideally:

- **Unbypassable**: mediates *all* TCB accesses
- **Tamper-resistant***:* cannot be sabotaged (by normal user or adversary). If it is, there is a fail-safe.
- **Verifiable**: "small enough to be subjected to analysis and tests" -- *[1]*



Reference Monitor: Concept

"In concept, the reference monitor mediates each reference made by each program in execution by checking the proposed access against a list of accesses authorized for that user."

"These principles…can result in integrating all of the system security controls for a system into one hopefully small portion of the operating system"

"Because the reference monitor concept implies interpretation of each reference made to determine the validity of the attempted access, efficient mechanisms for this interpretation are required if the concept is to be viable." -- *Computer Security Technology Planning Study [2]*

Reference Monitor: Race Conditions

Reference monitor checks to see if an action is allowed, *then* allows action to be performed. Attacker needs to change conditions *after* the check and *before* the use.

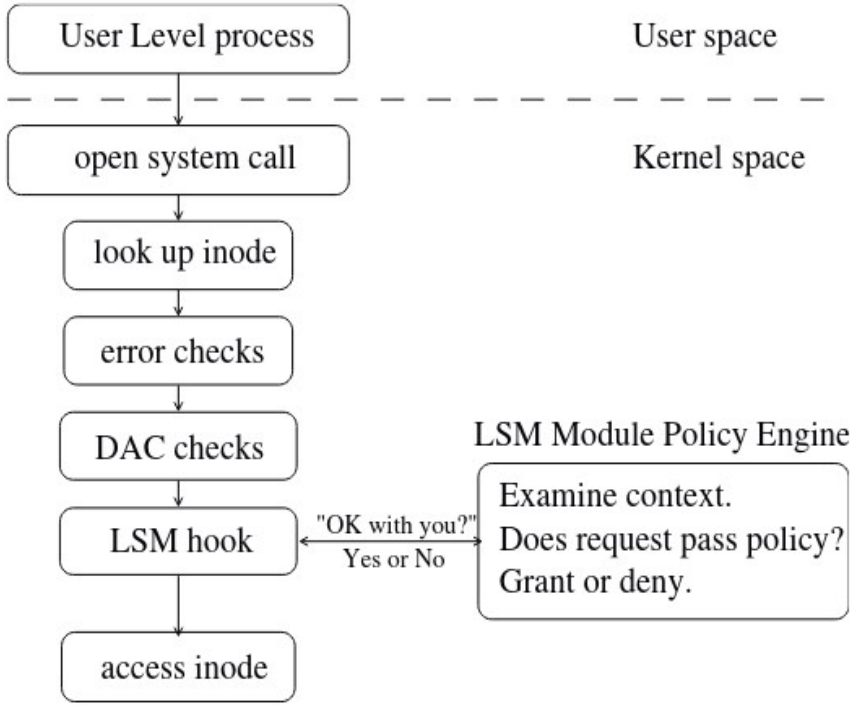
```
// check if user is allowed to write
to intended file
if (access("file", W_OK) != 0)
          exit(1);

// actually open /etc/passwd
f = open("file", O_WRONLY);
// overwrite passwords
write(f, buffer, sizeof(buffer));
```

```
// attacker waits for access check
// point file to /etc/passwd
symlink("/etc/passwd", "file");
// file opened
```

Linux Security Modules

LSM: reference monitor implementation to enforce mandatory access control.

- Insert hooks into kernel code just ahead of access (to deter ToCToU exploit)
- Hook calls LSM function to allow or deny access.
  - Discretionary or mandatory access control
- AppArmor: included in Linux kernel from 2.3.x
- SELinux: from 2.6.x



Reprinted from: Linux Security Modules: General Security Support for the Linux Kernel [7]

Access Control List

Map object to users and actions.

grades → [<100,1>, RW]
syllabus → [<100,1>, RW], [<200,2>, R]

An ACL allows the concept of a user **role**. Suppose the user with UID 200 is a student TA:
grades → [<100,1>, RW], **[<200, 1>, RW]**
syllabus → [<100,1>, RW], [<200,2>, R]

Easy to revoke access.

Capability List

Map user to objects and actions.

<100,1> → [grades, RW], [syllabus, RW]
<200, 2> → [syllabus, R]

Cryptographically protect from user tampering:

1. Client *C* requests server *S* to create object *O*
2. *S* creates *O* and random check *K*
3. *S* stores *K* in i-node with *O*
4. *S* sends capability [*S* id, *O* #, <rights>, *F*(*O*, <rights>, *K*)] to *C*
5. *C* requests access by sending capability
6. *S* verifies request using *K*

Difficult to revoke access. Easy for process encapsulation.

Bell-LaPadula Model

Objects and processes (users) have security levels (unclassified, confidential, secret, top secret)

Process at level k can:

- Read only objects at level $\leq$ k
- Write only objects at level $\geq$ k

Focus on **confidentiality**: no information can leak down from a higher level.

OS assigns users a level along with UID, GID.

Biba Model

Bell-Lapadula keeps secrets, but does not guarantee data **integrity**. Biba is the reverse.

Process at level k can:

- Write only objects at level $\leq$ k
- Read only objects at level $\geq$ k

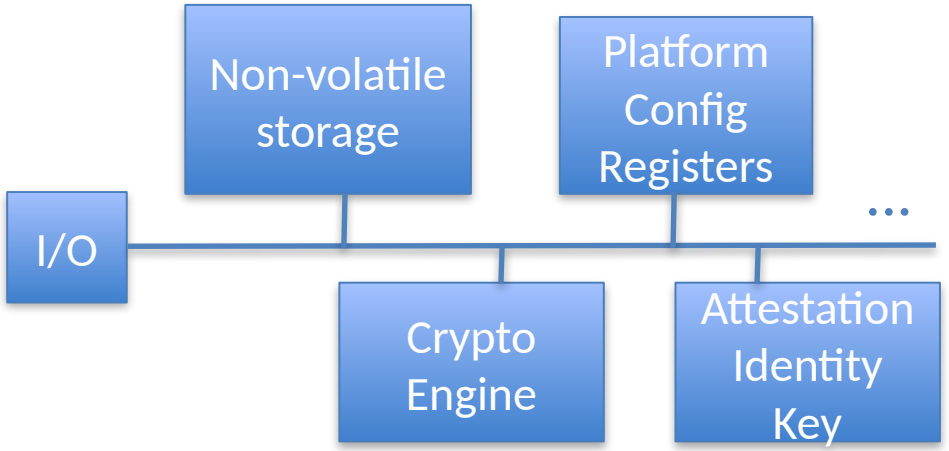
In practice, we have mixes of discretionary/mandatory access control and BLP/Biba.

Trusted Platform Module

How do we store cryptographic keys on a system that may not be secure?

Trusted Platform Module (TPM)

- Secure cryptoprocessor with integrated keys
- Encrypts keys and provides **attestation** of host state



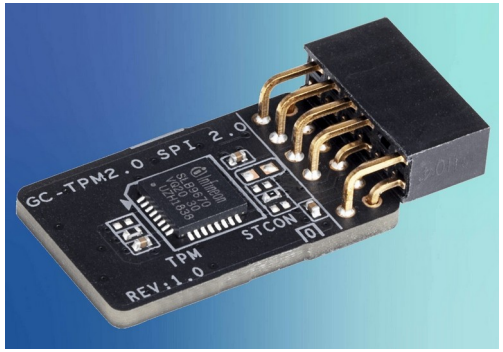
Components:

§ Non-volatile storage (integrated keys)

§ ≥ 16 20-byte **PCR**s (Platform Config Registers) to store integrity metrics

§ Crypto engine (RSA, SHA, RNG, signatures, …)

§ …



https://pcworld.com/article/394765/what-is-a-tpm-where-do-i-find-it-and-turn-it-on.html

TPM Keys

Storage Root Key (*SRK*)

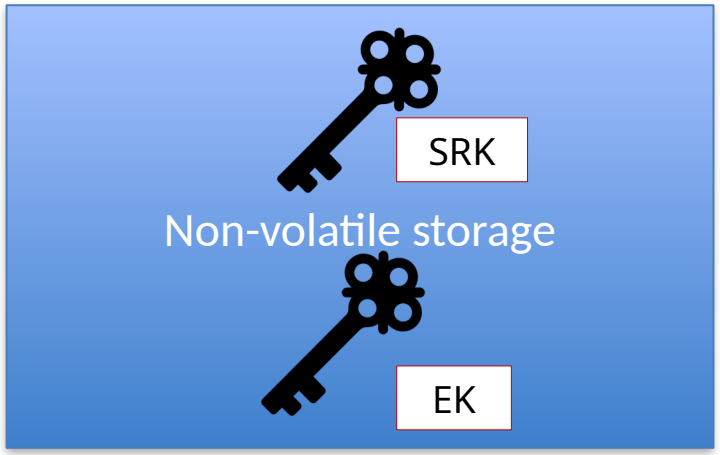
- RSA key-pair
- Encrypts external keys (sealing)
  - Root of Trust for Storage
- Decryption only if PCR values match

Endorsement Key (*EK*)

- Unique RSA key-pair
- EK_{sec} must never be disclosed
- Signed EK_{pub} proves TPM genuine for attestation

Attestation Identity Key(s) (*AIK*)

- Alias of EK_{sec}
- Used to sign PCR values for attestation
- Loaded into volatile storage



PCR Specification

PCR	Use	Notes
PCR0	Core System Firmware executable code (aka Firmware)	May change if you upgrade your UEFI
PCR1	Core System Firmware data (aka UEFI settings)	
PCR2	Extended or pluggable executable code	
PCR3	Extended or pluggable firmware data	Set during Boot Device Select UEFI boot phase
PCR4	Boot Manager Code and Boot Attempts	Measures the boot manager and the devices that the firmware tried to boot from
PCR5	Boot Manager Configuration and Data	Can measure configuration of boot loaders; includes the GPT Partition Table
PCR6	Resume from S4 and S5 Power State Events	
PCR7	Secure Boot State	
PCR8	Hash of the kernel command line	Supported by grub and systemd-boot
PCR 9	Hash of the initrd	Scheduled for linux v5.17
PCR10	Reserved for Future Use	
PCR11	BitLocker Access Control	
PCR12	Data events and highly volatile events	
PCR13	Boot Module Details	
PCR14	Boot Authorities	
PCR 15 to 23	Reserved for Future Use	

https://wiki.archlinux.org/title/Trusted_Platform_Module

TPM Specifications

TPM 1.2:

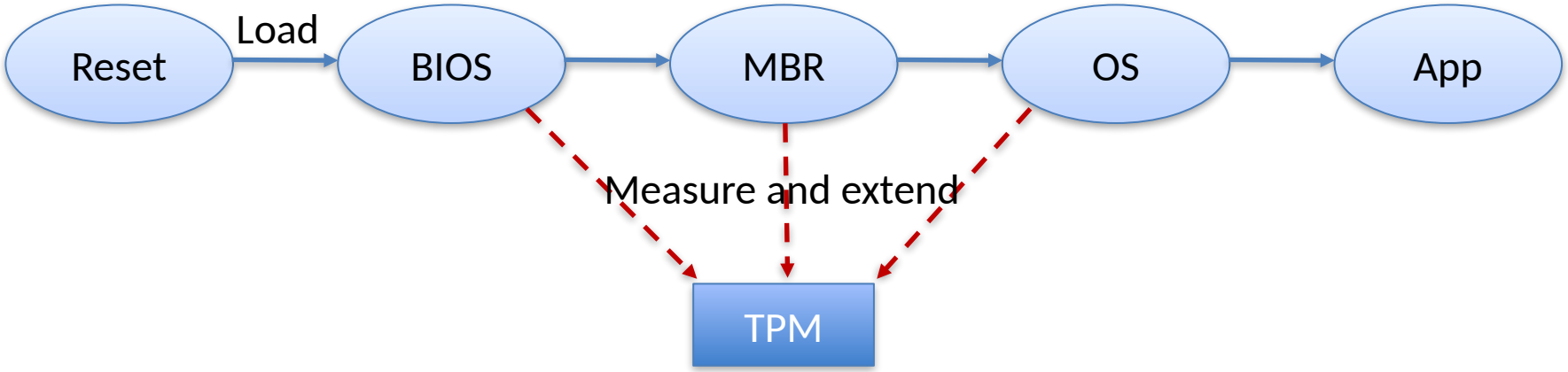
- RSA, AES-128, SHA-1 required
- One key (storage)
- Discrete chip

TPM 2.0:

- RSA, SHA-1, SHA-256, ECC, AES-128 required
- Multiple keys for storage and endorsement
- Discrete, firmware, hypervisor

https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/tpm-recommendations

Attestation Integrity Measure



Measure system configuration at load-time of each stage.

API: `Extend(n || digest)`: $PCR_n \leftarrow SHA(PCR_n \text{ || } digest)$

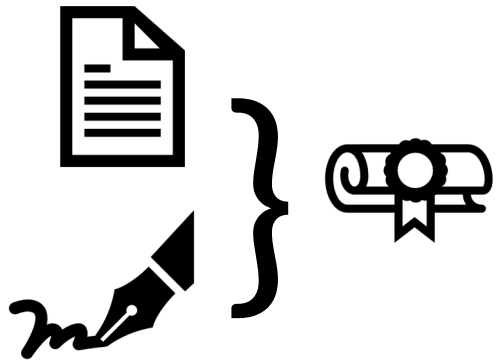
`Extend(n || <BIOS code>); Extend(n || <GRUB code>); ...`

Key unsealed by

1. first checking that the PCRs (states) match
2. then decrypting the sealed key

Digital Signatures

- KeyGen() → (*sec*, *pub*)
- $\sigma \leftarrow \text{Sign}_{sec}(msg)$
- $\text{Verify}_{pub}(msg, \sigma) \rightarrow 0/1$

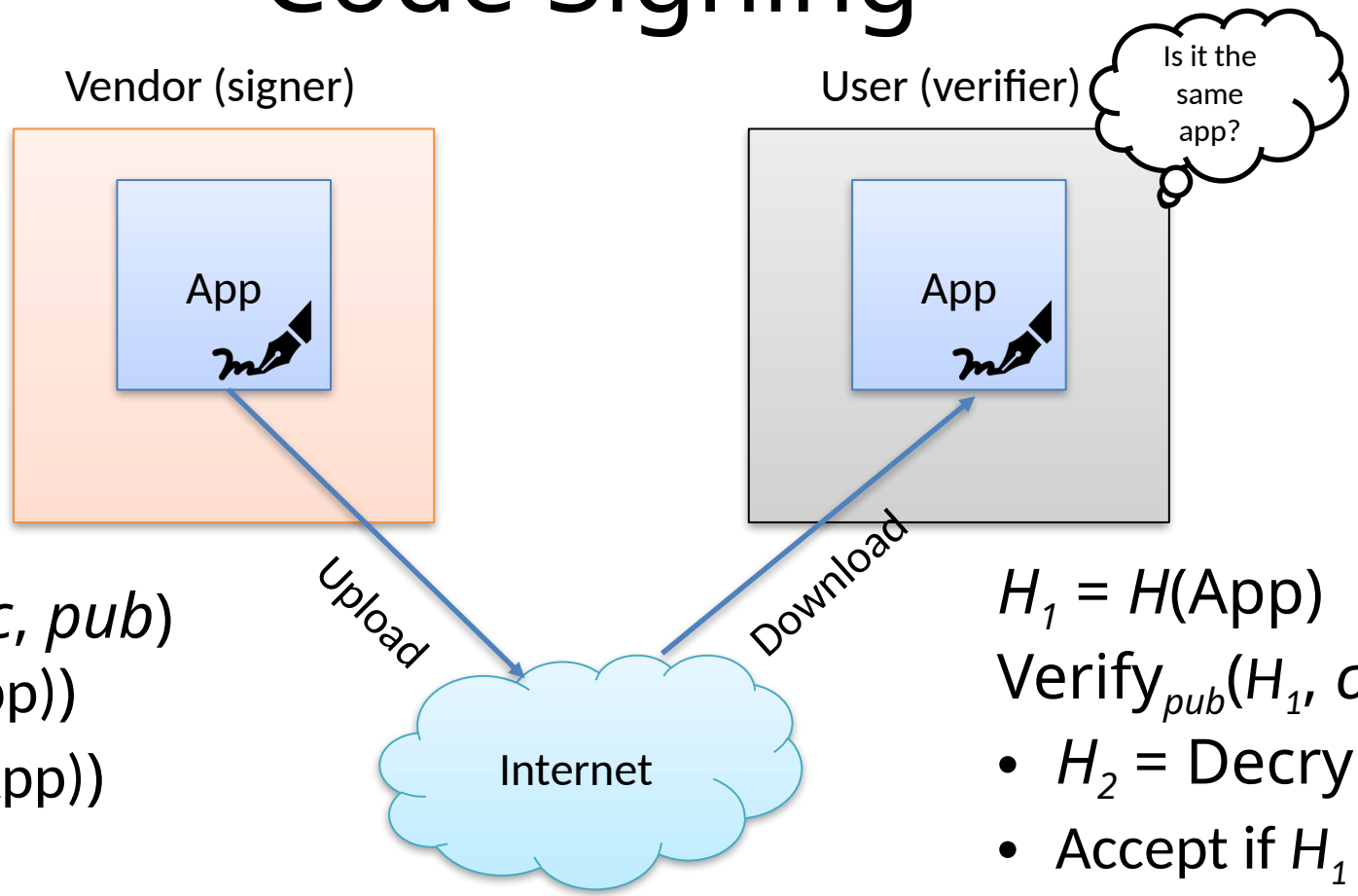


Certificate binds a party to a public key.

- Allows nonrepudiation
- An initial trusted party (CA) securely distributes its public key
- Subsequent signature forms a proof via certificate chain
- Trust $Alice_{pub}$ because we have $cert_{Bob \rightarrow Alice}$, Bob_{pub}, $cert_{Charlie \rightarrow Bob}$ and we inherently trust Charlie

TPM 2.0 supports ECDSA, ECSchnorr, ECDAA

Code Signing



KeyGen() → (*sec*, *pub*)

$\sigma \leftarrow \text{Sign}_{sec}(H(\text{App}))$

- $\text{Encrypt}_{sec}(H(\text{App}))$

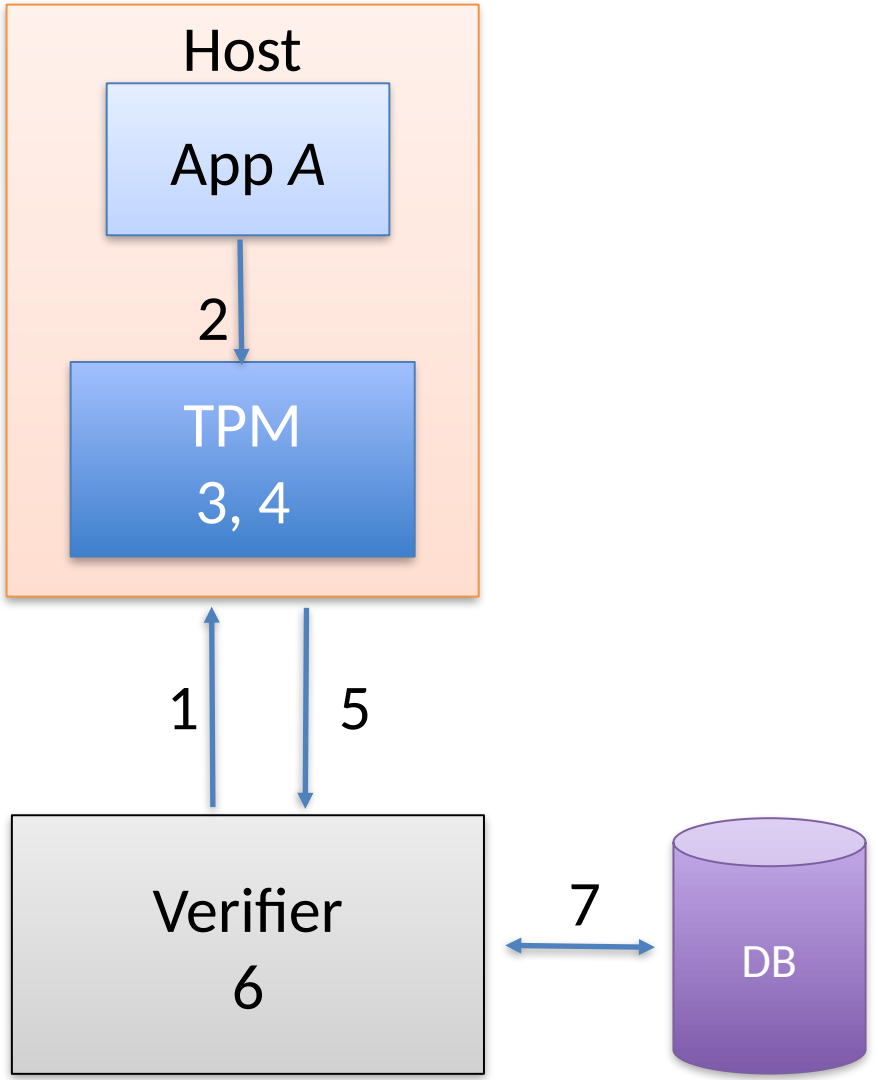
$H_1 = H(\text{App})$

$\text{Verify}_{pub}(H_1, \sigma)$

- $H_2 = \text{Decrypt}_{pub}(\sigma)$
- Accept if $H_1 == H_2$

Attestation Protocol

1. Verifier sends challenge on application *A*
2. TPM extends PCR values with hash *H* of host state (incl. *A*)
3. TPM obtains $C_1 = cert_{CA \rightarrow TPM}(AIK_{pub})$
4. TPM signs $C_2 = cert_{TPM \rightarrow A}(H)$ with AIK_{sec}
5. Host sends C_1, C_2
6. Verification of certificate chain
7. Verifier checks *H*



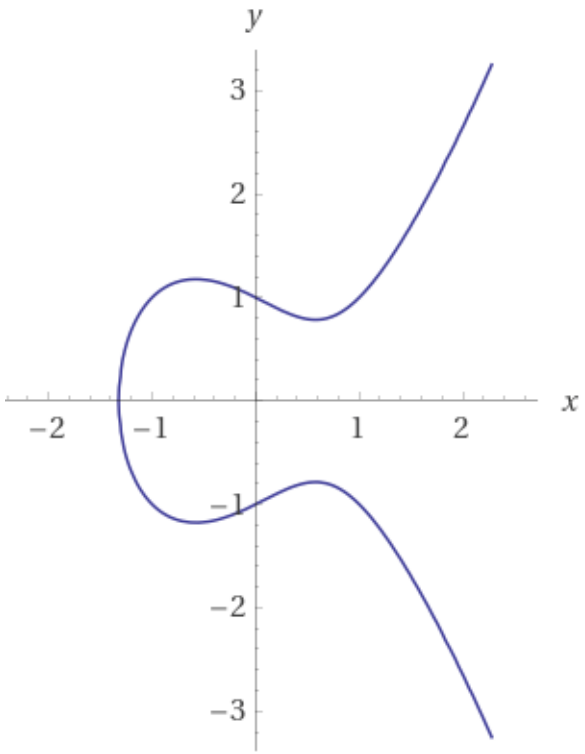
ECDSA: Elliptic Curves

Curve over a finite field with points satisfying

$$y^2 = x^3 + ax + b$$

Discrete logarithm problem:

- Given base point *B* and point *P*
- Infeasible to find *c* s.t. *P* = *c* * *B*
- For *N*-bit security, use field of order 2^{2N}



$y^2 = x^3 - x + 1$

ECDSA Signing

Alice wants to sign some message *m*. She does the following:

- Generates integer *sec*
- Sets *pub* = *sec* * *G*
  - *G* is a generator of (large) prime order subgroup
- Generates an integer **nonce** *k*
- Sets $r = (k * pub)_x$
- Sets $s = k^{-1}(H(m) + r * sec)$

Signature = (*r*, *s*)

Nonce must be secret and unique. Otherwise it reveals *sec*

Timing Leakage ("TPM-Fail")

Abstract

Trusted Platform Module (TPM) serves as a hardware-based root of trust that protects cryptographic keys from privileged system and physical adversaries. In this work, we perform a black-box timing analysis of TPM 2.0 devices deployed on commodity computers. Our analysis reveals that some of these devices feature secret-dependent execution times during signature generation based on elliptic curves. In particular, we discovered timing leakage on an Intel firmware-based TPM as well as a hardware TPM. We show how this information allows an attacker to apply lattice techniques to recover 256-bit private keys for ECDSA and ECSchnorr signatures. On Intel fTPM, our key recovery succeeds after about 1,300 observations and in less than two minutes. Similarly, we extract the private ECDSA key from a hardware TPM manufactured by STMicroelectronics, which is certified at Common Criteria (CC) EAL 4+, after fewer than 40,000 observations. We further highlight the impact of these vulnerabilities by demonstrating a remote attack against a StrongSwan IPsec VPN that uses a TPM to generate the digital signatures for authentication. In this attack, the remote client recovers the server's private authentication key by timing only 45,000 authentication handshakes via a network connection.

TPM-FAIL: TPM meets Timing and Lattice Attacks

Daniel Moghimi[1], Berk Sunar[1], Thomas Eisenbarth[1, 2], and Nadia Heninger[3]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2]University of Lübeck, Lübeck, Germany
[3]University of California, San Diego, CA, USA

The vulnerabilities we have uncovered emphasize the difficulty of correctly implementing known constant-time techniques, and show the importance of evolutionary testing and transparent evaluation of cryptographic implementations. Even certified devices that claim resistance against attacks require additional scrutiny by the community and industry, as we learn more about these attacks.

https://usenix.org/system/files/sec20-moghimi-tpm.pdf

TPM-Fail: CVEs

CVE-2019-11090: "Cryptographic timing conditions in the subsystem for Intel(R) PTT ... may allow an unauthenticated user to potentially enable information disclosure via network access."

CVE-2019-16863: "STMicroelectronics ST33TPHF2ESPI TPM devices before 2019-09-12 allow attackers to extract the ECDSA private key via a side-channel timing attack because ECDSA scalar multiplication is mishandled, aka TPM-FAIL."

TPM-Fail: Attack Phases

Phase 1 (generate profile):

- Attacker generates signatures and records timing information.
- Recovers nonces using known keys to find timing/nonce correlation.

Phase 2 (mount attack):

- Collects signatures (r_i, s_i) and timing samples t_i from vulnerable TPM implementation.
- Filters collected data, keeping signatures where bias in nonce k_i fits profile

Phase 3 (recover key):

- Recovers *sec* using lattice technique
  - LLL algorithm

TPM-Fail: ECDSA Nonce Leakage

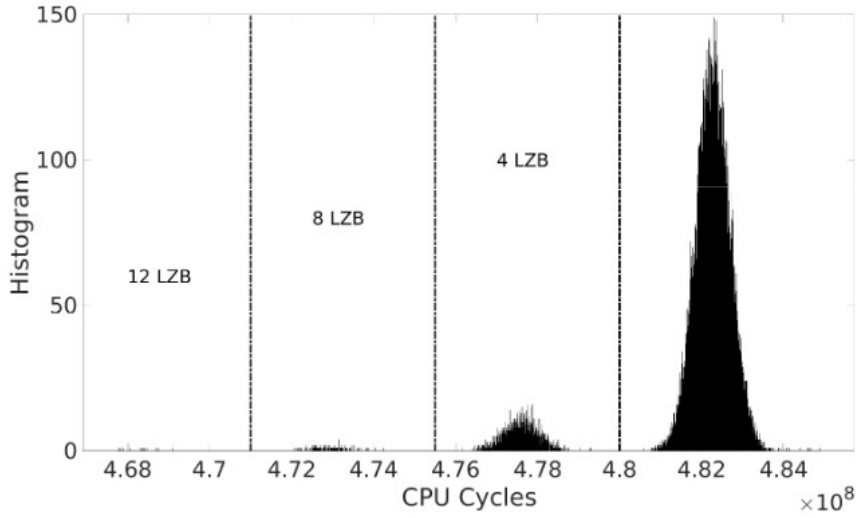


Figure 4: Histogram of ECDSA (NIST-256p) signature generation timings on Intel fTPM as measured on a Core i7-7700 machine for 40K observations.

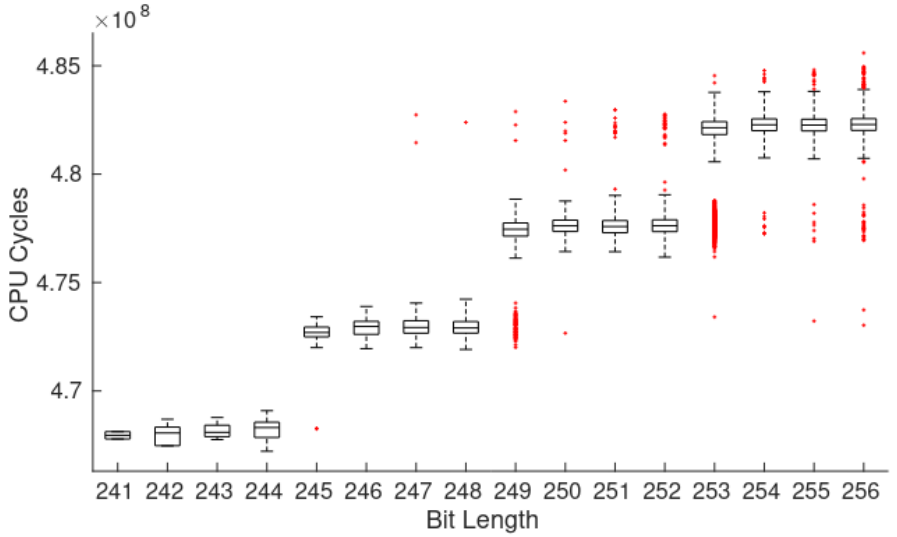


Figure 5: Box plot of ECDSA (NIST-256p) signature generation timings depending on the nonce bit length shows a clear step-wise relationship between the execution time and the bit length of the nonce for Intel fTPM.

Computing scalar multiplication for *r* value in signature proceeds window by window over nonce bits. Most significant window (MSW) bits are related to signature execution time.

Execution time leaks nonce information.

- 0 MSWs have faster execution time
- Intel PTT nonces have 4-bit MSW.
- Nonce is useful to extract *sec*.

TPM-Fail: Lattice Cryptanalysis

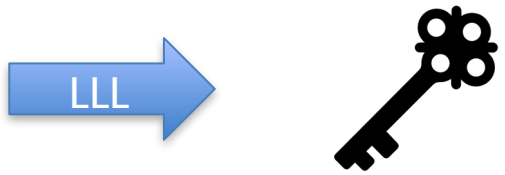
$s_i = k_i^{-1}(H(m_i) + d * r_i) \bmod n \rightarrow k_i - s_i^{-1} * r_i * d - s_i^{-1} * H(m_i) = 0 \bmod n$

$\rightarrow \mathbf{k_i} + A_i * \mathbf{d} + B_i = 0 \bmod n$

Let $K \geq k_i$ [Boneh and Venkatesan, 1996]

Shortest Vector Problem: find $(k_1, k_2, \ldots, k_t, K/n, K)$, hence d

$$\begin{bmatrix} n & & & & & \\ & n & & & & \\ & & \ddots & & & \\ & & & n & & \\ A_1 & A_2 & \ldots & A_t & K/n & \\ B_1 & B_2 & \ldots & B_t & & K \end{bmatrix}$$



References

[1] 5200.28-STD, DoD. 1985. Trusted Computer System Evaluation Criteria. Dod Computer Security Center.

[2] Anderson, James P. 1972. “Computer Security Technology Planning Study.” Vol. 2. U.S. Air Force Electronic Systems Division.

[3] Chen, Liqun. 2005. “Direct Anonymous Attestation (DAA).” Trusted Systems Laboratory, Hewlett Packard Laboratories, Bristol; https://trustedcomputinggroup.org/wp-content/uploads/051012_DAA-slides.pdf.

[4] Dworkin, M. 2016. "Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, Special Publication (NIST SP)." National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-38B.

[5] House of Representatives, Congress. "44 U.S.C. 3542 - Definitions". Government. U.S. Government Publishing Office, December 30, 2011. https://www.govinfo.gov/app/details/USCODE-2011-title44/USCODE-2011-title44-chap35-subchapIII-sec3542.

[6] Moghimi, Daniel, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2019. “TPM-FAIL: TPM Meets Timing and Lattice Attacks.” CoRR abs/1911.05673.

[7] NIST, and Emmanuel Aroms. 2012. NIST Special Publication 800-18 Revision 1 Guide for Developing Security

Plans for Federal Information Systems. CreateSpace

[8] Tanenbaum, Andrew Stuart, and Herbert Bos. 2015. Modern Operating Systems. Edited by Tracy Johnson. Fourth. Pearson.

[9] Wright, Chris, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. 2002. “Linux Security Modules: General Security Support for the Linux Kernel.” In USENIX Security Symposium, edited by Dan Boneh, 17–31. USENIX.